UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Máster en Investigación e Innovación en las Tecnologías de la Información y la Comunicación (Máster I^{2}TIC)

TRABAJO FIN DE MÁSTER

Aprendizaje por refuerzo y técnicas profundas aplicadas a un sistema de recomendación de venta al por menor.

Autor: Eloy Anguiano Batanero
Tutor: María Tsampa
Ponente: Pablo Castells Azpilicueta

septiembre 2019

Eloy Anguiano Batanero
Aprendizaje por refuerzo y técnicas profundas aplicadas a un sistema de recomendación de venta al por menor.

Eloy Anguiano Batanero

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mi familia y amigos.

¡Corre, plátano!

Ralph Wiggum

Agradecimientos

Me gustaría agradecer a *Accenture* y a su división de innovación *Accenture Digital* la oportunidad de poder formarme y experimentar con tecnologías tan interesantes y novedosas como son el *Deep Learning* y el *Deep Reinforcement Learning* en un equipo tan agradable. Dar especial gracias a Luis, Jesús, Paz y María por aceptarme sin problemas mi decisión de hacer el presente trabajo en la unidad de *Accenture Digital*; a Javi, Tamara, Azael y Sandra (y seguro que me dejo a alguien) por echarme la mano cuando lo necesitaba y responder mis constantes dudas; y a Sofía y Ruth por aguantarme (que es casi lo más difícil) y por esas tortillas de los viernes.

También dar las gracias a mi tutor, Pablo Castells, por no rechazar ninguno de mis asaltos a su despacho (que no hay otra forma de llamarlos y no han sido pocos) y por siempre atender a las dudas que tenía con una sonrisa.

Y por último, pero no por ello menos importante, gracias a mi familia y amigos, que son los que siempre me apoyan en los momentos en los que hace falta y los que se alegran en los buenos momentos.

Un abrazo a todos.

Resumen

Hoy en día, se ha podido comprobar cómo los modelos de *Deep Learning* y *Machine Learning* pueden ofrecer soluciones más óptimas a problemas tradicionales, y el caso de los sistemas de recomendación no es una excepción. Es por esto que *Accenture*, a través de su división de *Accenture Digital* realiza investigaciones periódicamente para mejorar cada uno de sus productos. Uno de estos productos es *ASI*, un sistema de recomendación orientado a la venta online al por menor o *retail* a través de la recomendación de anuncios en distintas plataformas *online*.

A lo largo de este *Trabajo Fin de Máster* se explorarán técnicas de *Deep Learning*/*Machine Learning* con el objetivo de observar qué estrategias podrían resultar más o menos provechosas a la hora de poner cada uno de los modelos en un entorno productivo.

Adicionalmente, se realizará una simulación de distintos agentes basados en *Deep Reinforcement Learning* para observar el rendimiento de cada una de las opciones así como las ventajas sobre los modelos anteriormente nombrados, ya que esos agentes son capaces de tener en cuenta recompensas más a largo plazo que las alternativas anteriores.

Palabras clave

Recuperación de Información, Sistemas de Recomendación, Aprendizaje Profundo, Aprendizaje Automático, Aprendizaje por Refuerzo, RecoGym, OpenAIGym

Abstract

Today, we have seen how *Deep Learning* and textitMachine Learning models can offer more optimal solutions to traditional problems, and the case of recommendation systems is no exception. That's why *Accenture*, through its division of *Accenture Digital* periodically conducts research to improve each of its products. One of these products is *ASI*, a recommendation system aimed at online retail sales or *retail* through the recommendation of advertisements on different *online* platforms.

Throughout this *Master's Final Thesis*, *Deep Learning*/*Machine Learning* techniques will be explored with the aim of observing which strategies could be more or less profitable when it comes to putting each of the models in a productive environment.

Additionally, a simulation of different agents based on *Deep Reinforcement Learning* will be made to observe the performance of each of the options as well as the advantages over the previously mentioned models, as these agents are capable of taking into account longer term rewards than the previous alternatives.

Keywords

Information Retrieval, Recommendation Systems, Deep Learning, Machine Learning, Reinforcement Learning, RecoGym, OpenAIGym

Índice

Listas

Lista de ecuaciones

Lista de figuras

Lista de tablas

1

Introducción

1.1. Objetivo

El objetivo de este *Trabajo de Fin de Máster* es explorar el rendimiento de distintas alternativas de algoritmos basados en *Deep Learning* y *Deep Reinforcement Learning* en el entorno de un sistema de recomendación orientado a la venta al por menor. Así:

- Se realizarán distintos procesados de datos para el entrenamiento de distintos modelos.
- Se propondrán distintos modelos de *Deep Learning* y se expondrán las ventajas y desventajas de unos y otros.
- Se estudiarán distintos agentes de *Deep Reinforcement Learning* a través de un entorno de simulación.

1.2. Motivación

Accenture Digital es la sección encargada de explorar los aspectos de carácter más innovador para los productos o servicios que pueda ofrecer *Accenture*. Uno de estos productos es *Accenture Smart Interactions* (ASI) , un sistema de recomendación orientado a la venta al por menor o *retail* a través de anuncios *online*. Por ello, *Accenture Digital* pretende siempre mantenerse en el uso y aplicación de las técnicas actuales en sus productos no sólo en este campo, sino en multitud de ellos. Estos campos siempre irán orientados a la digitalización, inteligencia artificial, *machine learning*, *big data*, etc.

Debido al carácter innovador que posee el grupo y teniendo un producto al que poder aplicarlo a corto plazo en cuanto se obtuviesen algunos resultados, era de gran interés estudiar el rendimiento de algunos algoritmos de *machine learning* y *deep learning* aplicados a un problema en concreto de recomendación. Una vez realizado dicho estudio, en el presente *Trabajo de Fin de Máster* se proponen de forma adicional alternativas basadas en el campo en reciente auge del *deep reinfordement learning* (más concretamente el *deep Q-learning*) permitiendo aprovechar la oportunidad de mejorar dicho sistema de recomendación maximizando ganancias más a largo plazo. La perspectiva del *reinforcement learning* es ineludible en un sistema de recomendación resulta muy interesante abordarlo ya que el

grueso de la literatura ve el problema como recomendación en un solo paso.

Se espera que este *Trabajo de Fin de Máster* sume resultados, así como una metodología de prueba para estos agentes y puedan compararse distintos algoritmos sin necesidad de llevarlos todos a producción para así seleccionar cuáles si sería interesante utilizarlos en el sistema real.

1.3. Organización del documento

El presente documento se organizará a través de los siguientes capítulos:

Capítulo 2: Sistemas de recomendación Explicación del funcionamiento y objetivos de un sistema de recomendación.

Capítulo 3: Aprendizaje profundo Explicación de aspectos y bases esenciales para el correcto entendimiento del funcionamiento del *deep learning*.

Capítulo 4: Aprendizaje por refuerzo Exposición del paradigma de aprendizaje por refuerzo, así como sus bases y alguna de sus alternativas.

Capítulo 5: Accenture Smart Interaction Presentación del producto de ASI , delimitación de las mejoras del producto a proponer y resultados de los experimentos realizados.

Capítulo 6: Conclusiones Conclusiones obtenidas a través del presente *Trabajo de Fin de Máster*, posibles líneas de trabajo y mejoras a futuro.

2

Sistemas de recomendación

2.1. Introducción

El campo de los sistemas de recomendación forma parte del campo de los sistemas de recuperación de información o *Information Retrieval*. Estos sistemas intentan ordenar una serie de ítems (películas, canciones, anuncios, etc.) en base a la posible relevancia que tendrán los ítems con los que aún no ha interactuado el usuario.

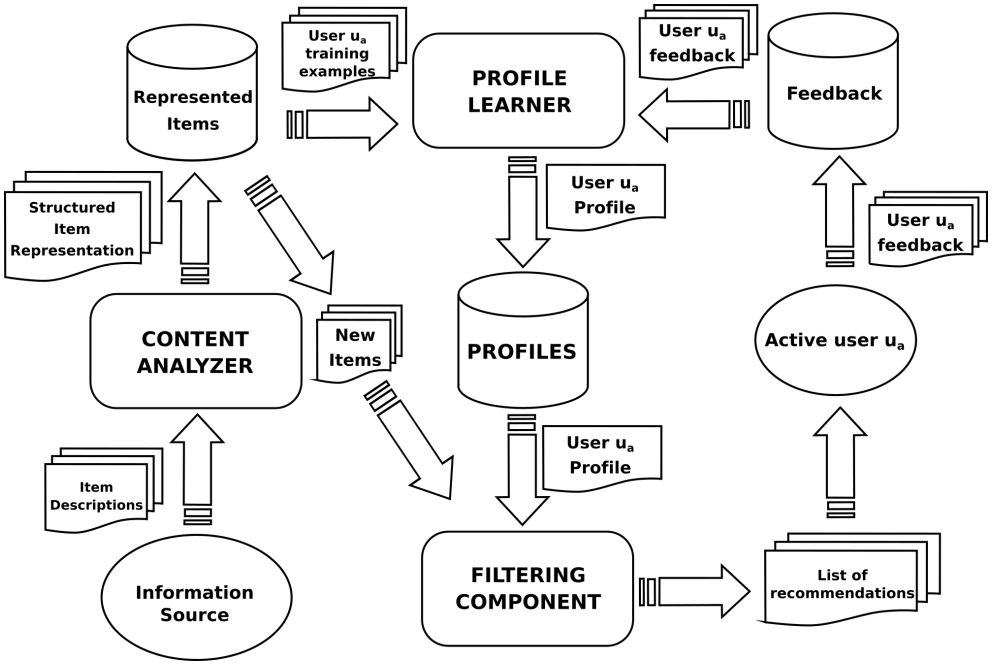


Figura 2.1: Imagen esquemática del cometido de un sistema de recomendación. Imagen extraída del libro “Semantics-Aware Content-Based Recommender Systems” [1]

Los sistemas de recomendación son ampliamente utilizados en plataformas web de entretenimiento como pueden ser *Spotify* a la hora de recomendar canciones o *Netflix* a la hora de recomendar películas, pero también existen otro tipo de sistemas no centrados en el entretenimiento del usuario que hacen uso de este tipo de técnicas, como puede ser *Amazon* al recomendarte compras basadas en tus compras anteriores o sistemas de selección de anuncios basados en la efectividad pasada que

tuvieron en el usuario y una larga lista de posibilidades.

Para llevar a cabo esta tarea existen una serie de algoritmos destinados a intentar predecir dicha relevancia, ya sea en base a la popularidad de los propios ítems en el sistema, en base a la interacción que ha tenido el usuario con ítems similares en el pasado, o utilizando la información de otros usuarios categorizados como similares al que se pretende recomendar.

2.2. Algoritmos tradicionales

2.2.1. Función similitud

Anteriormente hemos hablado de que algunos algoritmos de recomendación utilizan la información entre ítems o usuarios "similares". Para poder establecer esta comparativa de similitudes hemos de definir primero a qué nos referimos cuando hablamos de ítems o usuarios similares. Respondiendo a la necesidad de comparar elementos, nos encontramos con la función similitud.

La función similitud en sí misma puede ser constitutiva de un sistema de recomendación en sí mismo, pero también puede ser utilizada para sistemas de recomendación que la utilicen como componente para sus cálculos (como veremos más adelante). Existen multitud de funciones de similitud, pero en nuestro caso nos centratemos en la similitud coseno y la similitud de Jaccard.

Similitud Coseno

La similitud coseno es una métrica de similitud que se basa en proyectar cada elemento a comparar a un espacio N-dimensional para comparar el coseno del ángulo que forman los elementos proyectados entre sí. De esta forma, a menor ángulo entre las proyecciones de los elementos mayor similitud entre los mismos. La formulación matemática de ésta similitud es

$$cos(\theta) = \frac{A \cdot B}{|A|\,|B|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{2.1}$$

donde A_i y B_i son componentes de los vectores de las proyecciones de los elementos A y B, y donde $|A|$ y $|B|$ son los módulos de dichos vectores respectivamente. Al ser una métrica basada en el coseno, puede tomar valores en el rango $[-1, 1]$, siendo el valor -1 el peor valor de similitud posible y el 1 el mejor valor de similitud posible.

Como se puede observar en la figura 2.2, dicha métrica de similitud se encarga de medir la orientación de dichos vectores de proyección y no la magnitud de los mismos (al tratar con el ángulo entre ellos), por lo que resulta muy útil para encontrar elementos que tengan relaciones parecidas en sus coordenadas, sea cual fueren sus magnitudes.

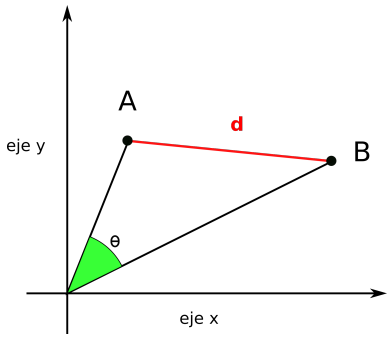


Figura 2.2: Imagen conceptual de la similitud coseno. Se presentan los distintos documentos a comparar como vectores multidimensionales (A y B) y la similitud coseno es inversamente proporcional al ángulo formado por ambos vectores.

Similitud de Jaccard

La similitud de Jaccard (también conocida como Índice de Jaccard) se trata de una métrica de similitud basada en álgebra de conjuntos. Mediante esta técnica se cuantifica cómo de coincidentes son dos conjuntos. Se define como

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{2.2}$$

donde los conjuntos A y B son cada uno de los elementos a comparar. Por ello, esta métrica toma valores en el rango $[0,1]$, siendo el 0 el valor para el cual los conjuntos son completamente disjuntos y el 1 para cuando tienen una coincidencia absoluta.

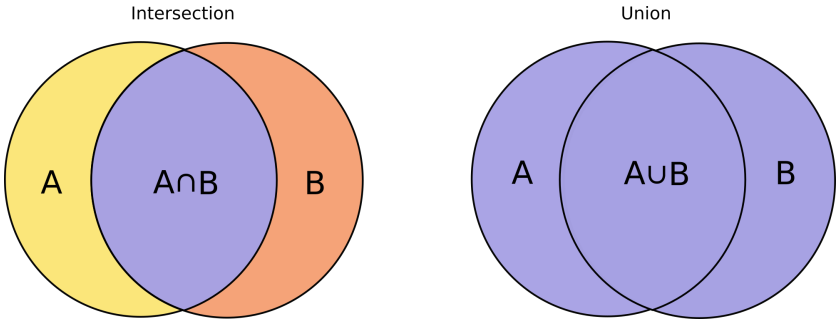


Figura 2.3: Imagen conceptual de la similitud de Jaccard. Este índice cuantifica la coincidencia entre dos conjuntos, siendo el valor 1 la coincidencia albsoluta y 0 la disjunción.

Esta similitud trata a los elementos a comparar como conjuntos algebraicos (figura 2.3), de forma que se busca que dichos elementos tengan componentes coincidentes, y se vuelve a huir de medir la magnitud de las mismas, como en la similitud coseno.

2.2.2. Algoritmos basados en contenido

Los algoritmos de recomendación basados en contenido tratan de recomendar los ítems con atributos similares, según la función similitud elegida (ver subsección 2.2.1), a los que han interactuado con el usuario de forma positiva.

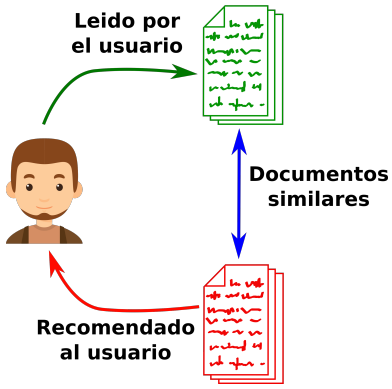


Figura 2.4: Imagen esquemática de un sistema de recomendación basado en contenido. En este tipo de sistemas se intentan recomendar ítems similares a los que han interactuado de forma positiva con el usuario anterioremente.

Este tipo de algoritmos resultan muy eficaces cuando se posee poca información del usuario, ya que desde el momento que interactúa con un sólo ítem del sistema ya se es capaz de realizar recomendaciones relativamente buenas. Además, estas recomendaciones son explicables por el sistema, es decir, al usuario se le pueden explicar las razones por las que se le recomienda determinado ítem (figura 2.4).

2.2.3. Algoritmos de filtrado colaborativo

Los algoritmos de filtrado colaborativo se basan en la premisa de que, al interactuar los usuarios con un determinado sistema, generan algún tipo de "Inteligencia colectiva". En otras palabras, entienden que los usuarios que han interactuado de forma similar (según la similitud explicada en el apartado 2.2.1) a otros usuarios se puede presuponer que interactuarán parecido en el resto de items o viceversa.

El algoritmo de filtrado colaborativo basado en usuario (homólogo al basado en ítems, por lo que sólo se explicará ésta opción) trata de recomendar a un usuario los ítems bien puntuados por usuarios similares a él. La formulación matemática de una de sus múltiples versiones (k Nearest Neighbours (kNN)) es

$$\hat{r}(u,x) = \frac{\sum_{v \in V_k(u)} sim(u,v) \cdot r(v,x)}{\sum_{v \in V_k(u)} |sim(u,v)|} \tag{2.3}$$

donde $\hat{r}(u,x)$ es la valoración que suponemos que el usuario u dará al ítem x con el que no ha

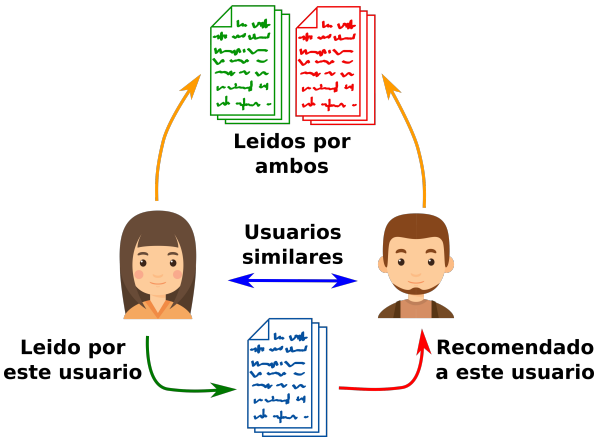


Figura 2.5: Imagen esquemática de un sistema de recomendación de filtrado colaborativo. Este tipo de sistemas intentan recomendar ítems que han interactuado positivamente con usuario similares entre ellos según sus interacciones pasadas.

interactuado, $V_k(u)$ es el conjunto de los k usuarios más similares al usuario u y $sim(u,v)$ es la función similitud aplicada al usuario u y v. Para ver este algoritmo de una forma más símple, se podría resumir en "usuarios que compraron el ítem i también compraron el ítem j".

Este tipo de algoritmos son especialmente sensibles al sesgo o *bias* que pueda tener un usuario en los sistemas en los que la relevancia se obtenga de manera explícita del mismo (un sistema de puntuación de 1 a 5, por ejemplo). Por ello, existen variantes que tienen en cuenta el sesgo de cada usuario según la relación entre la relevancia actual y la media de las pasadas. Estas se denominan versiones "centradas en la media". Una aproximación matemática sería

$$\hat{r}(u,x) = \bar{r}(u) + c \sum_{v \in V_k(u)} sim(u,v)(r(v,x) - \bar{r}(v)) \tag{2.4}$$

donde $\bar{r}(u)$ es la media de las valoraciones hechas por el usuario u en el sistema y c es una constante de normalización.

El mayor problema que poseen este tipo de algoritmos es el llamado *Cold Start*. Al basarse en el paradigma de "Inteligencia Colectiva" es capaz de hacer recomendaciones realmente buenas a medida que el usuario va interactuando cada vez más con el sistema, pero resulta bastante complicado hacerlas cuando no se dispone de esa información del usuario, por ejemplo, cuando el usuario comienza a utilizar el sistema.

Por ello, es bastante común que se utilice este tipo de algoritmo en combinación con alguna otra estrategia inicial (por ejemplo basado en contenido) para dar lugar a un sistema híbrido.

2.3. Aprendizaje automático en sistemas de recomendación

2.3.1. Factorización de matrices

La factorización de matrices es un algoritmo de recomendación de filtrado colaborativo que utiliza la descomposición de la matriz de valoraciones de los ítems por parte de los usuarios a un espacio de factores latentes. Dicha proyección de la matriz al espacio de factores latentes se hace a través de técnicas de aprendizaje automático, como por ejemplo utilizando una adaptación de Singular Value Decomposition (SVD) adaptado a matrices dispersas.

Figura 2.6: Imagen esquemática de un sistema de recomendación por factorización de matrices. Esta alternativa se basa en la descomposición de cada usuario y cada item en un vector de atributos latentes.

Para calcular las posibles futuras valoraciones que los usuarios le darán a un determinado ítem se hace de la forma

$$\hat{r}_{ui} = I_i^t \cdot U_u \tag{2.5}$$

donde I_i es el vector de factores latentes del ítem i, y U_u es el vector de factores latentes del usuario u. Estos vectores se calculan minimizando el error cuadrático medio

$$min_{I*,U*} \sum_{(u,i)} (r_{ui} - I_i^t \cdot U_u)^2 + \lambda(|I_i|^2 + |U_u|^2). \tag{2.6}$$

El parámetro λ es un parámetro que sirve para regularizar el aprendizaje y no sobreentrenar a los datos de entrenamiento.

2.3.2. Deep Learning

Actualmente, la tendencia de los sistemas de recomendación más novedosos es utilizar distintos modelos de aprendizaje profundos para realizar un *ranking* de los ítems que no han interactuado con el usuario. Estas técnicas pueden ser desde Perceptrones Multicapa y *Autoencoders* para optimizar una función de ajuste o modelos como Recursive Neural Nets (RNNs) o Convolutional Neural Nets

(CNNs) para ajustar dicha función tratándola en forma de serie temporal.

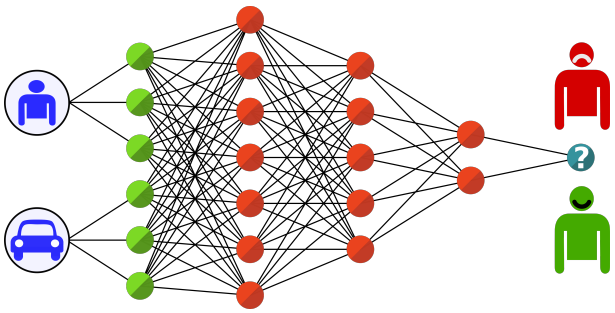


Figura 2.7: Modelos de Deep Learning aplicados a sistemas de recomendación. Este tipo de sistemas utilizan un modelo neuronal de múltiples capas para recomendar items a los usuarios.

Estos modelos han demostrado ser bastante eficaces, por lo que la tendencia de su uso es creciente en la literatura del estado del arte de los sistemas de recomendación [2].

2.4. Evaluación de los sistemas de recomendación

2.4.1. Feedback

Un sistema de recomendación trata de ordenar ciertos elementos o ítems en relación a la relevancia que tengan estos mismos para el usuario que utilice el sistema. Al ser el término "relevancia" uno con múltiples interpretaciones según el contexto, primero es necesario establecer cómo se va a decidir recopilar la información de dicha relevancia para el usuario. Para ello, existen dos formas de actuar:

Feedback explícito: Se le pide al usuario específicamente que valore cómo de adecuada es una determinada recomendación. Por ejemplo, con una valoración numérica del 1 al 5, puntuaciones positivas y negativas, etc. Este tipo de estrategias de recopilación de información son especialmente sensibles al sesgo del propio usuario, pero existen técnicas para lidiar con el efecto de los mismos como se ha comentado anteriormente en la sección 2.2.3.

Feedback implícito: En esta ocasión el usuario interactúa libremente con el sistema y se establecen sus interacciones como medidas de relevancia implícitas. Por ejemplo, si un usuario hace *click* con un ítem recomendado frente a otros puede entenderse que dicho elemento resulta más relevante que el resto mostrados o, en el caso de los entornos multimedia, podríamos hablar del tiempo de reproducción del archivo.

2.4.2. Evaluación Offline: Métricas

Al estar tratando con un concepto tan subjetivo y ambiguo como es la relevancia de un determinado ítem para un usuario, existen multitud de métricas con distintas interpretaciones y valores de lo que podría significar el término "relevancia".

Para poder aplicar este tipo de evaluación se divide el conjunto de valoraciones del sistema en dos subconjuntos, comúnmente denominados como *train* y *test*. Se utilizan los datos del subconjunto de *train* para intentar predecir las valoraciones de *test*. De esta forma se puede cuantificar cómo de acertada ha sido la predicción frente a la interacción real.

Métricas de error de predicción

Algunas de estas métricas se centran en cuantificar numéricamente cómo de acertada ha sido la predicción frente a la realidad estableciendo una medida de la distancia entre ambos valores. Algunas de las métricas de error de predicción más utilizadas son el Mean Absolute Error (MAE) , el Mean Squared Error (MSE) y el Root Mean Squared Error (RMSE) :

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|t_i - y_i| \qquad MSE = \frac{1}{N}\sum_{i=1}^{N}(t_i - y_i)^2 \qquad RMSE = \frac{1}{N}\sqrt{\sum_{i=1}^{N}(t_i - y_i)^2}. \tag{2.7}$$

Métricas de ranking

Muchas veces no resulta tan interesante ajustarse lo máximo posible a la valoración real, sino a la ordenación según la definición de relevancia establecida. Las métricas de *ranking* establecen cómo de ajustada es la ordenación resultante de la predicción frente a la real, olvidándose de las magnitudes de dichas predicciones. Algunas de las métricas más utilizadas en éste ámbito se basan en álgebra de conjuntos, como pueden ser

$$Precision = \frac{|Rel \cap Ret|}{|Ret|} \qquad Recall = \frac{|Rel \cap Ret|}{|Rel|} \qquad F = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}. \tag{2.8}$$

Rel es el conjunto de elementos calificados como relevantes para el usuario y Ret es el conjunto de elementos devueltos por el sistema.

Por otro lado, existen métricas capaces de discriminar dicha ordenación según distintos grados de relevancia como Normalized Discounted Cumulative Gain (nDCG)

$$nDCG = \frac{DCG}{IDCG} \qquad DCG = \sum_{i=1}^{Ret}\frac{2^{rel_i} - 1}{log_2(i+1)} \qquad IDCG = \sum_{i=1}^{Rel}\frac{2^{rel_i} - 1}{log_2(i+1)} \tag{2.9}$$

donde DCG cuantifica la ordenación según los grados de relevancia del conjunto de elementos retornados del sistema en cuestión mientras que $IDCG$ representa la ordenación ideal según la relevancia de los elementos retornables.

Métricas adicionales

Ya que el problema de la recomendación de elementos a un usuario se trata de un problema bastante ambíguo tanto en la definición de qué es relevante para un usuario como en qué recomendaciones aportan valor al usuario y cuáles no, hay múltiples métricas dependiendo del problema a tratar y las necesidades del sistema.

Diversidad y Novedad

Existen métricas dedicadas a cuantificar distintos tipos de escenarios que intentan maximizar la ganancia de información para el usuario final también basadas en *ranking* y que intentan aumentar la variabilidad de interacción de usuarios con ítems. Ejemplos de este tipo de métricas pueden ser "Diversidad" o "Novedad" [3].

Mientras que la métrica de "Novedad" se refiere a cómo de diferente es un elemento con respecto visualizaciones previas por un determinado grupo o colectivo, "Diversidad" indica cómo de diferentes son una serie de elementos entre sí.

Click-through rate

Otra métrica interesante (sobre todo para sistemas de recomendación orientados a la visualización de anuncios) [4] es el Click-through rate (CTR) . El CTR se define como

$$CTR = \frac{|Clicks|}{|Impressions|} \cdot 100 \tag{2.10}$$

donde $|Clicks|$ es el número de *click-throughs* de un elemento y $|Impressions|$ es el número de impresiones del mismo. De esta forma, tenemos una efectividad porcentual de la conversión de las impresiones de elementos recomendables a *clicks* por los usuarios de determinado sistema. Pese a que esta métrica se utiliza también para la evaluación *online*, es posible simularla de forma *offline*.

2.4.3. Evaluación Online: Tests A/B

En contraposición a las métricas de evaluación anteriormente comentadas en el apartado 2.4.2, existen formas de comprobar en vivo el rendimiento de cada uno de los sistemas de recomendación propuestos. Este tipo de estrategias se denominan tests A/B.

Los tests A/B se basan en desviar tráfico destinado al sistema A hacia el sistema B (figura 2.8) y cuantificar la efectividad que tiene el sistema B en relación a la cantidad de tráfico desviado. Así se puede comprobar en vivo si el sistema B es capaz de rendir de una forma más efectiva que el sistema A.

Para atribuir la recomendación convertida (supongamos que la conversión se corresponde con

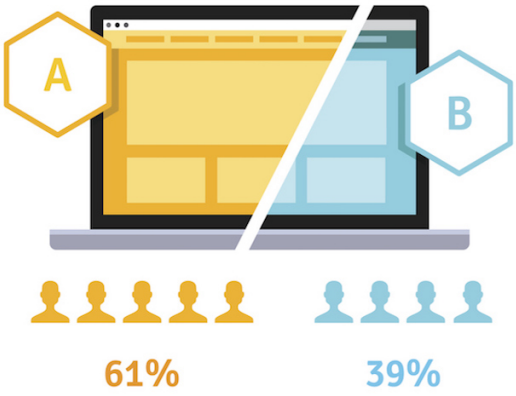


Figura 2.8: Imagen esquemática del funcionamiento de test A/B. En la figura, el sistema A se lleva el 61 % del tráfico, mientras que al sistema B se le asigna el 39 % del tráfico pudiéndose así comparar ambas alternativas Imagen extraída de TrustRadius [5]

un *click* por parte del usuario) a cada uno de los sistemas que se quiera evaluar en vivo existen distintas opciones en base a cómo se haya formado el *ranking* resultante combinado de cada uno de los sistemas. Algunas de las alternativas posibles son:

Método equilibrado: Tras calcular todas las permutaciones posibles de *clicks* a los ítems presentados, se comparan los *rankings* generados por ambos sistemas y se otorga un punto al *ranking* que hubiese sido capaz de generar de forma más eficaz cada una de las secuencias de *clicks*.

Método Team Draft: Se calculan todos los *rankings* posibles combinando los generados por el sistema A y el sistema B intercalando los elementos entre ambos sin repetición. Cuando el usuario hace *click* en uno de los elementos del *ranking* combinado y se le atribuye al *ranking* que ha generado en cada ocasión dicho elemento.

Método probabilistico: Cada elemento tiene una probabilidad asignada de ser generado por cada uno de los *rankings* de los sistemas a evaluar. Posteriormente se calcula la probabilidad conjunta de elegir dicho elemento de cada uno de los *rankings*.

3

Aprendizaje profundo

3.1. Redes neuronales

Las redes neuronales, el algoritmo de predicción que usaremos más adelante, son un método de resolución de problemas "bioinspirado". Es importante conocer las bases biológicas sobre las que se apoya el paradigma antes de tratar la abstracción utilizada en el mundo de la informática teórica. Por esta razón, procederé a la explicación de las bases de las neuronas biológicas y sus componentes para identificar las partes de una neurona artificial y qué función le compete a cada una de ellas en el modelo original.

3.1.1. La neurona biológica

En 1906, concedieron a Ramón y Cajal y Golgi el Premio Nobel de Medicina y Fisiología por sus trabajos de investigación de la estructura del sistema nervioso. Golgi y Cajal defendían hipótesis opuestas sobre la organización del sistema nervioso. Ramón y Cajal propuso la *"Doctrina de la Neurona"* [6], según la cual se establece la neurona como célula básica de operación en el sistema nervioso y, por tanto, de las redes neuronales biológicas.

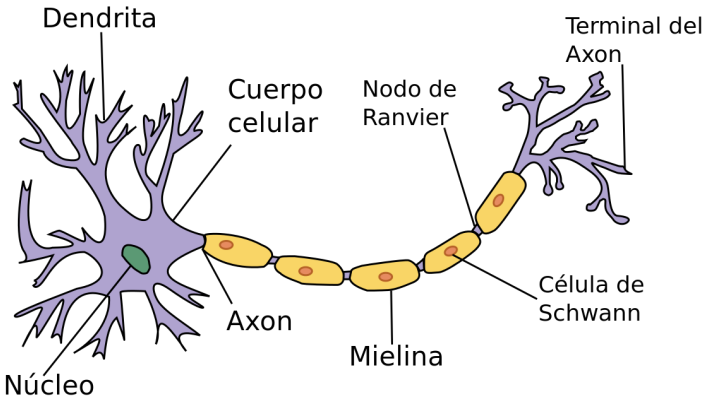


Figura 3.1: Imagen esquemática de una neurona biológica. (Imagen extraída de Wikipedia [7]).

Las neuronas establecen circuitos de potenciales eléctricos a través de la excitabilidad de sus membranas. Esta excitabilidad depende de la conductancia establecida por los puentes sinápticos de dichas neuronas, las cuales pueden servir de entrada a otras neuronas y así se irá conformando una red neuronal.

Pese a que existan distintos tipos de neuronas biológicas (según el propósito que tengan las mismas), todas comparten una serie de elementos comunes.

Sinápsis: Es un proceso de conexión entre dos neuronas mediante el cual una libera cierto tipo de neurotransmisores con el objetivo de producir una reacción a las neuronas colindantes. Esta liberación de neurotransmisores hará que la conexión sináptica entre dos neuronas determinadas pueda verse reforzada o mermada según interese.

Dendritas: Son las prolongaciones de la neurona post-sináptica encargadas de recibir los estímulos de neurotransmisores producidos por distintas neuronas pre-sinápticas.

Soma: Se trata del cuerpo celular de la neurona. Su membrana es una capa lipídica que es capaz de propagar potenciales eléctricos según lo recibido a través de las dendritas.

Axón: Se trata de la otra prolongación de la parte distal de la neurona por la cual se liberan los neurotransmisores al espacio sináptico para ser recogidas por las nuevas neuronas post-sinápticas.

En el cerebro humano hay del orden de miles de millones de neuronas con un promedio de 7000 conexiones por neurona. Las conexiones y topología de las redes conformadas por esas neuronas nos permiten desarrollarnos y aprender del mundo que nos rodea.

3.1.2. La neurona artificial

En 1943, Warren Sturgis McCulloch y Walter Harry Pitts propusieron un modelo de neurona artificial (la neurona de McCulloch-Pitts) [8] para simular el comportamiento de la neurona biológica, solo que estableciendo un modelo matemático más simple.

En la figura 3.2 vemos que existen las mismas partes de la neurona biológica simplificadas o modeladas lo suficientemente funcionales para la aproximación en cuestión pero no con la complejidad del sistema biológico real. Homólogamente a la neurona biológica, en la neurona artificial tenemos:

Vector de pesos de conexiones (Sinapsis): Valores que simbolizan los pesos de las conexiones entre neuronas de la forma $W = \{W_{11}, W_{12}, ...W_{ij}\}$ donde W_{ij} es el peso entre la neurona pre-sináptica i y la neurona post-sináptica j.

Vector de valores de entrada (Dendritas): Colección de valores que corresponden a los valores de entrada de las dendritas de la neurona $X = X_i$.

Función de propagación (Soma): Función que aúna la información recibida por sus dendritas para que el resultado de esa con el valor umbral del potencial eléctrico de la membrana, o simplemente el valor umbral (w_0) en el modelo artificial, pueda ser tratado posteriormente por la función de activación.

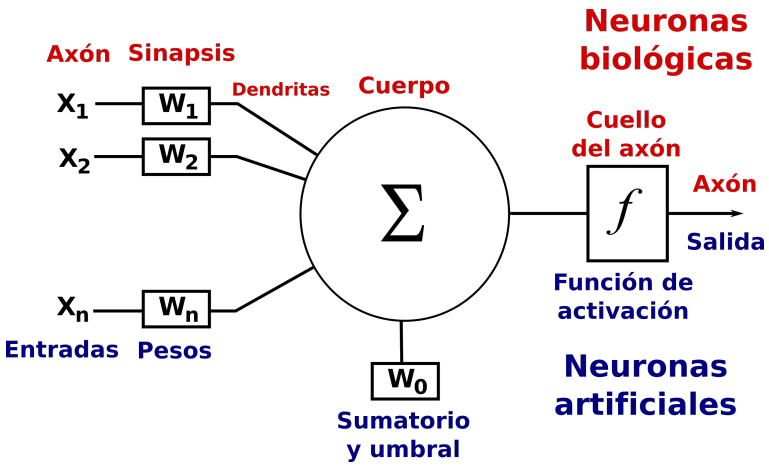


Figura 3.2: Comparación entre una neurona biológica y una artificial. Imagen extraída del Blog de Fernando Sancho Caparrini [9]).

$$y_j = w_0 + \sum_{i=1}^{N} x_i * w_{ij}. \tag{3.1}$$

Función de activación (Axón) Es la que se encarga de establecer la cuantía de excitación que emitirá la neurona pre-sináptica hacia las neuronas post-sinápticas. Utilizando la función de propagación anterior, tendríamos:

$$f(y_j) = f(w_0 + \sum_{i=1}^{N} x_i * w_{ij}). \tag{3.2}$$

En el apartado **??** se tratará una de estas funciones en específico, ya que se trata de una función clave para el aprendizaje profundo.

3.2. Aprendizaje en las redes neuronales

El aprendizaje de una red neuronal se basa en actualizar los pesos entre las sucesivas capas de la red neuronal para minimizar la función de error. La minimización de la función de error será calculada a través del descenso por gradiente. Por ello, para cada capa, la actualización de pesos W_n se calculará de la siguiente forma sucesiva

$$W_n = W_{n-1} - \lambda \overrightarrow{\nabla E} \tag{3.3}$$

siendo λ la tasa de aprendizaje, es decir, la magnitud del paso que se da en la dirección indicada por el gradiente del error $\overrightarrow{\nabla E}$ calculado mediante el algoritmo de *backpropagation* que se explicará en el apartado 3.2.1.

3.2.1. Algoritmo de backpropagation

El algoritmo de *backpropagation* o retropropagación [10] es el algoritmo que nos permite calcular el gradiente del error capa a capa de la red neuronal ($\overrightarrow{\nabla E}$). Actualizar todos los pesos de la misma para minimizar esa función de error es lo que se entiende como "aprender".

Este algoritmo se basa en que, partiendo de un error obtenido con la comparación de la salida obtenida y y el valor objetivo t se pueda establecer el error de cada neurona de capas anteriores. Así se averigua la fracción de error provocada por cada camino individual desde la capa de salida hasta la entrada y se consigue actualizar los pesos minimizando la función de error elegida.

El algoritmo de retropropagación se encarga de calcular ese gradiente del error ($\overrightarrow{\nabla E}$) desde la capa de salida hacia la capa de entrada pero, al ir siempre acompañado de evaluar la salida actual (paso anterior de *feedforward*) y de actualizar los pesos (paso posterior), habrá casos en los que nos podremos referir a éste como un algoritmo de aprendizaje en general.

3.2.2. Optimizador ADAM

En 2015 Diederik Kingma y Jimmy Ba presentaron *ADAptative Moment estimation* (ADAM) . ADAM fue presentado en un artículo científico llamado "Adam: A Method for Stochastic Optimization" [11]. Este algoritmo combina de las fortalezas entre dos algoritmos anteriores: *Adaptative Gradient Algorithm* (AdaGrad) y *Root Mean Square Propagation* (RMSProp) .

ADAM propone una novedad al descenso por gradiente: el momento. El momento responde a la aceleración del descenso por gradiente. La expresión de la actualización de los pesos según este optimizador sería

$$W_n = W_{n-1} + \eta \Delta W_{n-1} - \lambda \overrightarrow{\nabla E}, \tag{3.4}$$

siendo $\Delta W_{n-1} = W_{n-1} - W_{n-2}$ y η la tasa del momento que le da importancia al cambio de pesos en la iteración anterior.

- β_1 Tasa de decrecimiento exponencial del gradiente para la estimación del momento.
- β_2 Tasa de decrecimiento exponencial del gradiente al cuadrado para la estimación del momento.
- λ Tasa de aprendizaje o *Step size*.
- ϵ Constante de prevención de la división por cero.

Este optimizador resulta clave para ayudar a la convergencia de redes neuronales con un gran número de capas (el denominado *deep learning*) hacia un mínimo local.

3.2.3. Activación ReLU

Como hemos visto en la sección 3.1.2, cada neurona necesita de una función de activación que unificará las entradas ponderadas de la misma. Al realizarse el aprendizaje de las redes neuronales por el algoritmo de *backpropagation*3.2.1 que minimiza el error a través de un descenso por gradiente, es interesante que dicha función sea sencillamente derivable.

A medida que se ha ido aumentando el número de capas en los modelos de regresión profundos (de ahí su nombre), se comprobó que las funciones de activación al uso, como podrían ser la función sigmoidal o tangente hiperbólica, sufrían del problema denominado *Vanishing Gradients Problem* o "Problema de disipación de los gradientes", para lo cual se propone la función de activación Rectified Linear Unit (ReLU) [12].

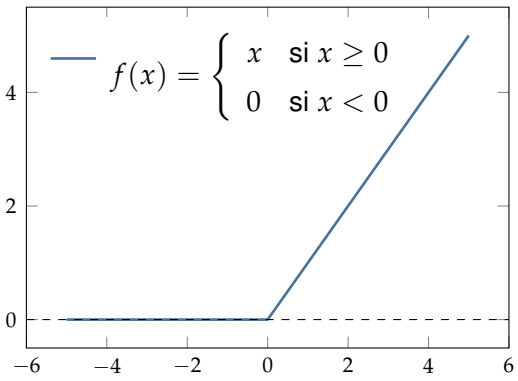


Figura 3.3: Representación gráfica y analítica de función lineal rectificadora ReLU

La función de activación ReLU , al tener una derivada inmediata que se define de la forma

$$\frac{df(x)}{dx} = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases} \tag{3.5}$$

reduce el tiempo de entrenamiento del descenso por gradiente y propone una solución al *Vanishing Gradients Problem* o "Problema de disipación de los gradientes" anteriormente comentado.

3.2.4. Inicialización de Xabier Glorot

En su artículo *"Understanding the difficulty of training deep feedforward neural networks"* [13], Xavier Glorot y Yoshua Bengio presentan su método particular de inicialización de pesos de una red neuronal. Su método se basa en conseguir establecer una normalización de los pesos y con ellos una forma de Gaussiana normalizada que permite que la actualización de pesos se realice en todas las capas, ya que un gradiente de error nulo significa que ese peso no será actualizado.

Así, se evitará un efecto de cuello de botella en la actualización de unas capas en detrimento de otras y nos permitirá actualizar de manera uniforme todas y cada una de las capas.

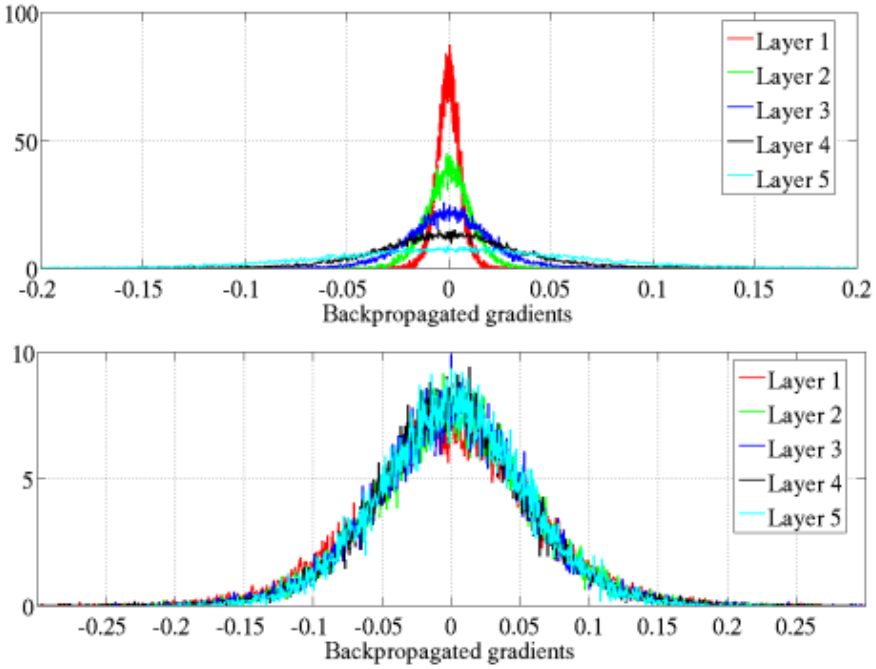


Figura 3.4: Histograma de gradientes por capa tras distintas inicializaciones. Inicialización heurística aleatoria (imagen superior) vs. Inicialización de Xavier (imagen inferior). Imagen extraída de la explicación del artículo [14].

3.3. Regularización

Las redes neuronales son una herramienta muy potente para encontrar una función de regresión que modele los datos presentados pero ¿y si son demasiado potentes?. Aquí es donde aparece el problema del sobreajuste u *overfitting*. El sobreajuste es el efecto de ajustar demasiado tu función a los datos de entrenamiento presentados, lo cual puede extraer relaciones no causales que merman la capacidad de generalizar de la red neuronal.

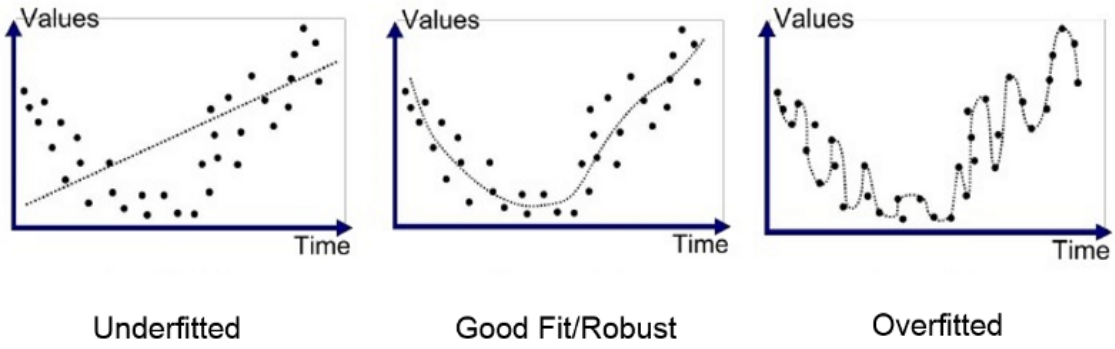


Figura 3.5: Ejemplo de *underfitting* (imagen de la izquierda) y *overfitting* (imagen de la derecha) en comparativa con un modelo correctamente ajustado (imagen central). Imagen extraída de Foro sobre *Machine Learning* [15].

Una red neuronal ha de ser capaz también de generalizar lo bastante como para encontrar la

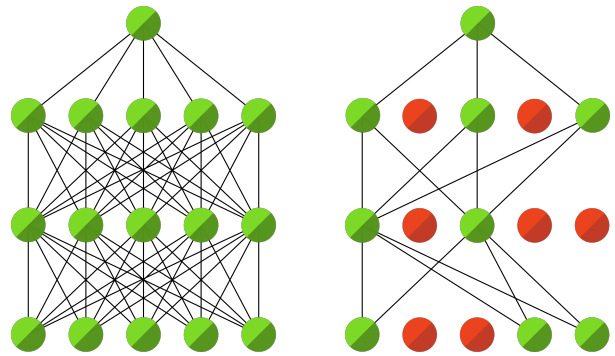
función más sencilla que modele lo suficientemente bien los datos presentados. Por ello, hemos de establecer una penalización al crecimiento de los coeficientes de la función de ajuste para alcanzar el equilibrio y encontrar una función de ajuste apropiada para el problema en cuestión y capaz de generalizar.

Para asegurar dicha generalización, existen métodos para favorecer la simplicidad de los pesos de la red neuronal. Un ejemplo de uno de estos métodos se denomina *L2 penalty*. A través de dicho método se le añadirá una penalización al resultado de la función de error de la propia red neuronal con mayores coeficientes en la función de ajuste:

$$E_{L2} = Error + \alpha \|W\|^2, \tag{3.6}$$

donde $\|W\|$ es la norma de la matriz de pesos W y α es la tasa de penalización que se quiere aplicar.

Sin embargo, una vez ilustrada la intencionalidad y necesidad de los métodos de regularización a la hora de entrenar una red neuronal, podemos pasar a plantearnos distintos métodos encaminados al mismo fin. Uno de éstos métodos, disponible a través de la librería *keras*, será la introducción de la capa de *Dropout* [16] al modelo.



(a) Red neuronal estándar. (b) Tras aplicar *dropout*

Figura 3.6: Imagen esquemática de capa de Dropout de una red neuronal [17]. En ella podemos ver en la figura 3.6(a) una red neuronal al uso, mientras que en la figura 3.6(b) podemos ver una fase de *feedforward* en una red con *dropout*.

Dicha capa se tendrá en cuenta únicamente durante la fase de entrenamiento y con una probabilidad p no se utilizará cada una de las conexiones hacia futuras capas para favorecer la elasticidad, robustez y capacidad de generalización de la red neuronal que se está entrenando.

3.4. Embeddings

Los *Embeddings* son un tipo de aprendizaje representacional a través del cual se realizan proyecciones de identificadores a un espacio a través de vectores latentes. En el campo del procesamiento

del lenguaje natural resultan de gran ayuda ya que el resultado de la proyección de las palabras (dichos identificadores) da lugar a un espacio en el que términos con usos parecidos serán cercanos unos a otros en el espacio resultante, como puede comprobarse en la figura 3.7.

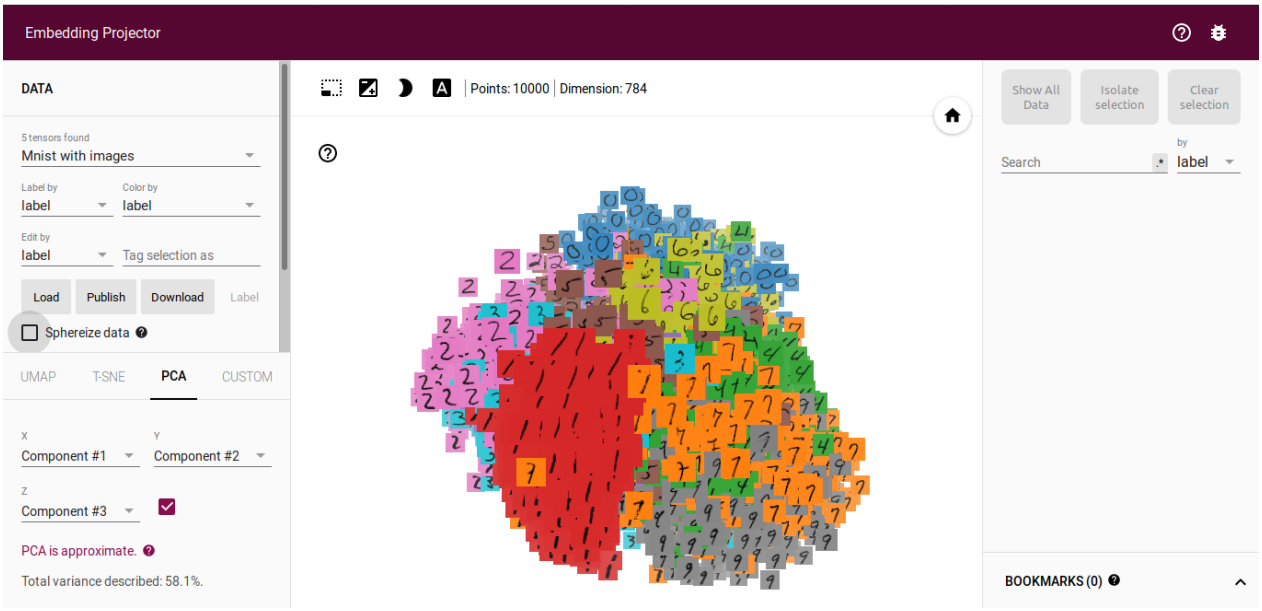


Figura 3.7: Representación de números manuscritos a través de un embedding generados a través de los datos de MNIST. En él se pueden ver cómo los números manuscritos se agrupan en el espacio resultante. Imágen extraida de la web de Embedding Projector

Dicha representación resulta ser una factorización de matrices implícita [18], por lo que al poseer la información de interacción entre usuarios e ítems propia de un sistema de recomendación, podemos utilizar este tipo de proyecciones para intentar acercar tanto usuarios semejantes en su propio espacio de proyección como ítems en el suyo.

4

Aprendizaje por refuerzo

4.1. Explicación del paradigma

En 1938, el psicólogo y filósofo Burrhus F. Skinner estudió los paradigmas del condicionamiento operante y el aprendizaje por refuerzo a través del experimento denominado como la “Caja de Skinner”.

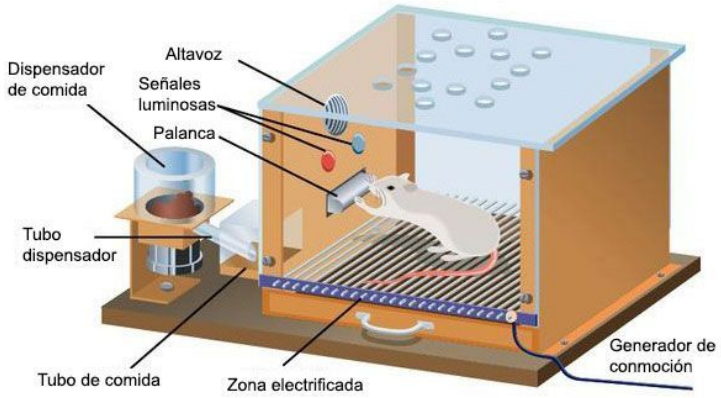


Figura 4.1: Imagen esquemática de la Caja de Skinner. En la caja hay una rata que puede realizar acciones que se verán recompensadas con comida o acciones que serán castigadas mediante electroestimulación. Imagen extraída del Blog de Psicoactiva [19]

A través de dicho experimento, Skinner pudo comprobar que determinados comportamientos de aprendizaje de los animales seguían un proceso iterativo de observación, acción y recompensa. En concreto, la cámara de condicionamiento operante recompensaba positivamente con comida al animal que realizaba la acción deseada, como accionar un botón o una palanca (figura 4.1).

Tomando como referencia este experimento, en el campo de las ciencias de la computación se realiza una aproximación con el fin de entrenar agentes inteligentes para resolver determinado tipo de problemas. Esta aproximación nace con el nombre de aprendizaje por refuerzo, conformando una alternativa a los algoritmos de *Machine Learning* de aprendizaje supervisado y aprendizaje no supervisado.

En la figura 4.2 podemos observar cómo la abstracción de este comportamiento psicológico se

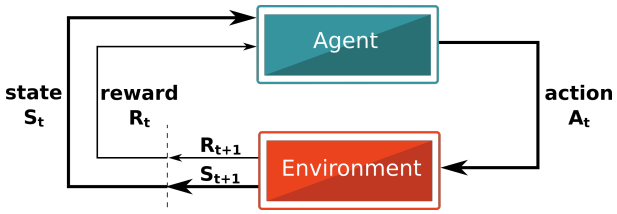


Figura 4.2: Imagen esquemática del proceso de aprendizaje por refuerzo. Se puede ver el proceso iterativo de estados, acciones y recompensas entre el agente y el entorno. Imagen extraída de KDnuggets [20]

compone de los siguientes conjuntos algebraicos:

S: Conjunto de posibles estados del sistema.

A: Conjunto de acciones que puede realizar el agente.

R: Conjunto de recompensas otorgables al agente.

A través de esta nomenclatura, los elementos S_t, A_t y R_t conformarían el estado, acción y recompensa a tiempo t del proceso iterativo de aprendizaje que siguen este tipo de agentes.

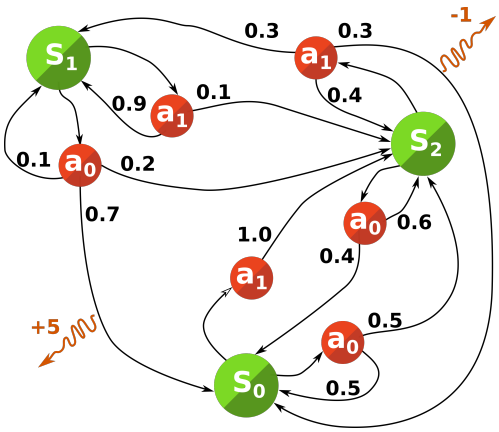


Figura 4.3: Imagen esquemática de un proceso de decisión de Markov. Se puede observar los distintos estados junto con las acciones y las probabilidades de transición entre los estados tras realizar cada una de las acciones correspondientes.

Matemáticamente, este tipo de problemas se pueden ver como un "Proceso de decisión de Markov" (figura 4.3). Los *Markov Decission Processes* (MDPs) son un tipo de modelado matemático para la toma de decisiones realmente útiles a la hora de estudiar problemas de optimización basados en dicho paradigma. La cuestión de este tipo de modelado matemático es encontrar la política de toma de decisiones o *policy* que maximice el retorno de la recompensa definida.

Debido a que este tipo de aprendizaje necesita de la interacción continua con un entorno que le proporciona la consecución de acciones y estados, es muy común ver cómo se hacen las distintas pruebas contra distintos videojuegos debido a la gran facilidad de reejecución y definición de estados, acciones y recompensas.

4.2. Agentes multibrazo

Uno de los algoritmos de aprendizaje por refuerzo más básicos se denomina agente multibrazo. Este tipo de algoritmos toma su nombre del problema que intenta resolver: el problema del bandido multibrazo [21].

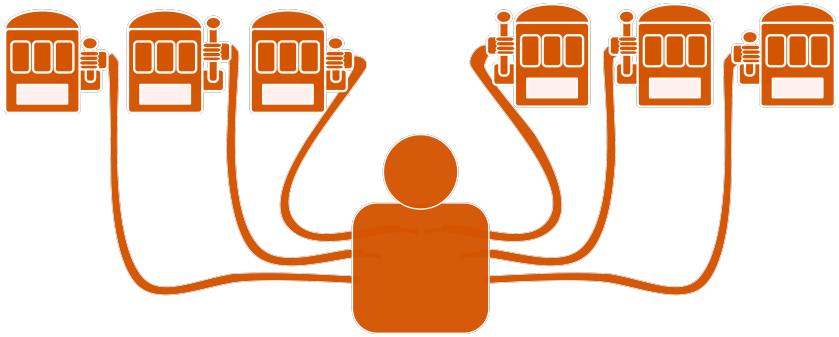


Figura 4.4: Imagen conceptual del problema del bandido multibrazo. Cada una de las máquinas tragaperras representa cada una de las posibleas acciones realizables por el agente. Cada máquina tendrá un retorno de inversión distinto y el agente determinará cuál es la de mayor recompensa.

Imaginemos que hay una serie de máquinas tragaperras, cada una de ellas con una distribución de probabilidad de éxito distinta. El cometido del agente es intentar maximizar las probabilidades de ganar el premio minimizando la inversión necesaria para hacerlo. El agente deberá ir explorando las distintas posibilidades de las que dispone, o conjunto de acciones A para accionar cada una de las tragaperras, e ir aprendiendo de las experiencias generadas para ser cada vez más óptimo a la hora de escoger en cuál de las máquinas debería invertir.

La dicotomía entre explorar o explotar la solución que se cree óptima en cada momento se trata de un problema abierto en sí y un punto crucial para la convergencia a la solución óptima de este tipo de algoritmos. Por ello, se tratará este problema más a fondo en el apartado 4.4, así como cada una de las alternativas más comunes.

4.3. Q-Learning

En muchas ocasiones, la acción que resulta óptima en el siguiente paso del proceso iterativo puede resultar no ser una buena acción si hablamos de obtener un buen rendimiento a largo plazo. Pensemos en una partida de ajedrez; *a priori*, realizar un movimiento que permita al oponente eliminar a la reina puede parecer una decisión nefasta, pero es posible que esta decisión nos permita ganar la partida observando la acción desde un punto de vista menos cortoplacista.

Para intentar suplir esta carencia que presentan los bandidos multibrazo explicados en el apartado 4.2, se introduce la función acción-valor $Q(a, s)$. La función acción-valor se define de la forma

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi \left[r + \gamma Q_\pi(s_{t+1}, a_{t+1}) \right] \tag{4.1}$$

donde π es la política de acciones actual, r la recompensa del paso actual y γ un factor de descuento (típicamente alrededor de $0{,}99$).

En el caso del *Q-Learning* el objetivo es conseguir establecer un equilibrio entre la obtención de recompensas en el corto y el largo plazo. De ello se encarga el factor de descuento de posibles recompensas futuras γ. Para realizar dicho cometido, el *Q-Learning* clásico elabora una tabla denominada *Q-table* de tamaño $S \times A$, donde S representa el conjunto de estados posibles al que se puede ver sometido el agente y A el conjunto de acciones que puede realizar el mismo. Dicha tabla se irá actualizando a medida que el agente se encuentre en cada una de las combinaciones de la misma, bien sea a través de la fase de exploración o la de explotación del mismo.

4.3.1. Deep Q-Learning

A medida que el conjunto de estados S y el de acciones A crecen, se hace necesario prescindir de la anteriormente denominada *Q-table* y es necesario sustituirla por algún tipo de estimador. Como se ha podido comprobar en el capítulo 3.1, una red neuronal profunda se trata nada más y nada menos que de un estimador realmente potente, por lo que podremos realizar la sustitución de dicha tabla por una red neuronal (*Deep Q Network* (DQN)) que estime el valor de la función Q para cada una de las posibles acciones $a \in A$ dado un estado $s \in S$.

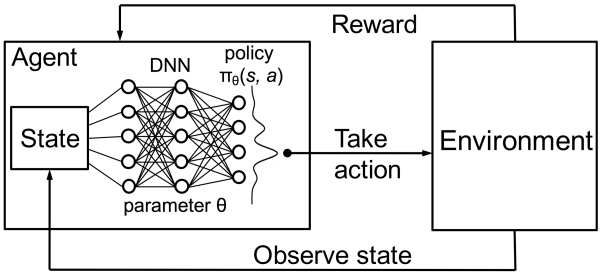


Figura 4.5: Diagrama esquemático de actuación de un agente basado en DQN. Imagen extraída del artículo “Resource Management with Deep Reinforcement Learning” [22]

De esta forma, se convierte el problema de la elección de una acción por el agente, en un problema de aprendizaje supervisado de clasificación. Se seleccionará como acción siguiente a tomar aquella que se estime tenga mayor valor de la función $Q(a, s)$.

Double Deep Q-Learning

Con la sustitución de la tabla de valores de la función $Q(a, s)$ (*Q-table*) por un estimador tan potente como una red neuronal (DQN) aparecen una serie de problemas contingentes que habrán de ser tenidos en cuenta.

Uno de esos problemas era la sobrestimacion de los valores de la función $Q(a,s)$. Como se ha expuesto en la ecuación 4.1, la elección de una acción dependerá de la esperanza de la ganancia a futuro de las siguientes acciones para tomar. El problema reside en que la fiabilidad de la aproximación correcta de la función $Q(a,s)$ puede ser muy ruidosa en las primeras etapas del entrenamiento y dependerá en gran medida de los estados y acciones explorados, lo que puede sesgar al agente hacia políticas no óptimas y complicar dicho entrenamiento.

La solución propuesta por *Hado van Hasselt* denominada *Deep Double Q-Learning* [23] propone disociar la selección de la acción de la estimación del valor de la función $Q(a,s)$ en dos redes distintas. Una de dichas redes seleccionará la acción de acuerdo a aquella que tenga el mayor valor de la función $Q(a,s)$, mientras que la otra se encargará de calcular el mismo valor de selección de acción en el siguiente estado. Matemáticamente (y de forma más simplificada), podemos reformular la ecuación 4.1 de la forma

$$Q(a_t,s_t) = r(s_t,a_t) + \gamma Q(argmax_{a_{t+1}} Q(a_{t+1},s_{t+1}),s_{t+1}) \tag{4.2}$$

donde una de las redes se encargará de estimar el valor de $argmax_{a_{t+1}} Q(a_{t+1},s_{t+1}),s_{t+1}$ mientras que la otra utilizará dicho valor para estimar $Q(argmax_{a_{t+1}} Q(a_{t+1},s_{t+1}),s_{t+1})$.

En el propio artículo en el que *Hado van Hasselt* propone esta solución se puede observar como incrementa los resultados de la estrategia DQN en la gran mayoría de los casos probados (figura del anexo A).

Dueling Deep Q-Learning

En esta ocasión, se intentará resolver la cuestión: ¿Por qué calcular el valor de todas las acciones dado un estado si se es capaz de determinar que un estado es intrínsecamente malo? Por ejemplo, y manteniendo el contexto de los videojuegos, un estado que lleve a un estado final negativo como la muerte sea cual sea la acción tomada.

Por ello, la propuesta vuelve a ser separar la decisión de la acción en dos estimadores o redes neuronales distintas [24]. En esta ocasión, una se encargará de estimar el valor del propio estado $V(s)$, mientras que la otra se encargará de estimar el valor de una acción en dicho estado $A(s,a)$. Matemáticamente, la función $Q(a,s)$ se estimará de la forma

$$Q(a_t,s_t) = A(s_t,a_t) + V(s_t). \tag{4.3}$$

4.3.2. Métodos Actor-Critic

Anteriormente, hemos tratado con métodos de aprendizaje con refuerzo únicamente basados en la estimación de los valores de la función $Q(a, s)$, ya bien sea por el método clásico de la *Q-table* o por estimadores basados en redes neuronales. Sin embargo, existen métodos que combinan estas estrategias con métodos que directamente intentan optimizar la política de decisiones (como puede ser el algoritmo *REINFORCE*, no tratado en el presente documento) llamados *Policy Gradient Methods*, los cuales utilizan directamente la recompensa total de los episodios.

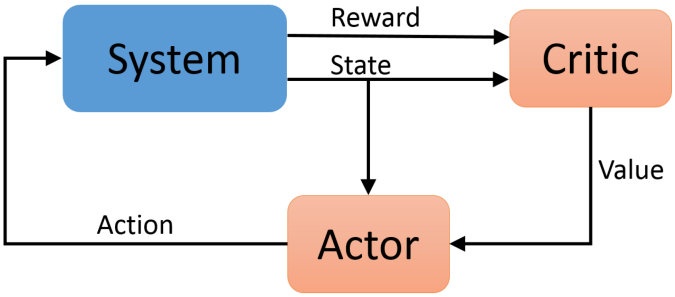


Figura 4.6: Imagen esquemática de agente basado en Actor Critic. Imagen extraída de artículo "A Long-Short Term Memory Recurrent Neural Network Based Reinforcement Learning Controller for Office Heating Ventilation and Air Conditioning Systems" [25]

El método *Actor-Critic* (y sus múltiples variantes) vuelve a utilizar dos redes neuronales para dos cometidos completamente distintos, una con el rol de actor y otra con el rol de crítico. En la figura 4.6 podemos ver que tendremos una red con el rol de crítico que se encargará de estimar el valor $Q(a, s)$ de ejecutar una acción a en un estado s, mientras que la red con el rol de actor se encargará de optimizar la política de decisiones del agente utilizando los valores de la función Q calculada por el crítico siguiendo las estrategias utilizadas en los *Policy Gradient Methods*.

4.4. Exploración VS Explotación

El proceso iterativo por el cual aprenden los agentes basados en aprendizaje por refuerzo tiene dos formas de escoger una acción a realizar en el siguiente paso: la exploración o la explotación.

Por un lado, la fase de explotación se encarga de ejecutar aquella acción que el agente cree óptima en el momento en el que se encuentra. Esta fase le servirá al agente para afianzar la política actual del mismo a la vez que podrá ir obteniendo las recompensas deseadas a la hora de diseñar el agente.

Por otro lado, tenemos la fase de exploración. Esta fase es esencial y problemática a partes iguales. Es esencial en tanto en cuanto el agente debe explorar las distintas posibilidades de ejecución de acciones ante los estados proporcionados por el entorno de manera que el agente sea capaz de aprender a reaccionar de una manera si no óptima, al menos aceptable. Sin embargo, resulta una fase

un tanto problemática ya que todas las veces que se decida explorar significa que no se está realizando la fase de explotación, lo que puede incurrir en que el rendimiento del agente sea nefasto.

Una de las fortalezas del aprendizaje por refuerzo reside en mantener continuamente una incertidumbre sobre si las acciones tomadas por el agente resultan ser óptimas. Por ello, para lidiar con el dilema entre ejecutar una fase de exploración o una de explotación existen una serie de algoritmos que veremos a continuación, cada uno con sus propias ventajas e inconvenientes que resultarán más o menos adecuados según las características del problema a resolver.

E - Greedy

El algoritmo ϵ-*Greedy* se basa en establecer una probabilidad ϵ de realizar una fase de exploración y una probabilidad de $(1 - \epsilon)$ de ejecutar una fase de explotación.

A pesar de que este valor de ϵ, que controla la proporción de ejecuciones que se destinarán a explorar en lugar de explotar, puede ser un valor fijo durante toda la vida del agente, es realmente interesante establecerlo como un valor variable. Es por esto que este algoritmo también hace hincapié en la paulatina actualización del valor de ϵ.

Es fácil suponer que las fases de explotación del agente más tempranas no optimizarán la ganancia de recompensas como lo harán fases más avanzadas. Por ello, se suele utilizar una tasa de decadencia para el valor ϵ hasta llegar a un mínimo no nulo del mismo destinado a garantizar la completa exploración a tiempo infinito.

Upper Confidence Bound

El algoritmo *Upper Confidence Bound* (UCB) o "Límite de cota superior" se basa en asumir que las ramas no exploradas del algoritmo resultan más ventajosas.

Para ello, a cada acción $a \in A$ a escoger se le asigna una recompensa esperada extremadamente alta. A medida que obtengamos las recompensas reales de cada una de las acciones a se irá promediando dicha recompensa esperada a través de las obtenidas. En cada iteración, el algoritmo escogerá la acción que posea el máximo valor esperado de recompensa.

Este algoritmo garantiza que todas las acciones serán exploradas al menos una vez. Sin embargo, asume que las recompensas de algunas acciones puedan cambiar a lo largo de toda la vida del agente, es decir, en el caso en que el algoritmo ha conseguido converger a que una de las acciones resulta la más ventajosa, nunca ejecutará una nueva exploración hacia una de las acciones con menos valor de recompensa esperada, a diferencia del algoritmo ϵ-*Greedy* explicado en el apartado 4.4.

4.5. Memorias

Una vez explicados los conceptos de cómo los agentes basados en *reinforcement learning* obtienen las experiencias de las que aprender, nos hemos de enfrentar a otro problema.

Necesitamos que nuestro agente sea capaz de reaccionar frente a situaciones "muy conocidas" o comunes de una forma correcta, por lo que se hace necesario añadir un elemento que nos ayudará al aprendizaje del agente: una memoria.

La memoria de un agente de este tipo guarda en cada una de sus entradas las experiencias pasadas: el estado inicial, la acción realizada, la recompensa obtenida, etc. Esta memoria permite que en cada interacción del agente, al ir a entrenar con la nueva experiencia que acaba de experimentar se le añada una fase en la que entrenará con las experiencias acumuladas en la memoria denominada "*experience replay*".

4.5.1. Memoria secuencial

Una de las alternativas más sencillas y con buenos resultados es utilizar una memoria secuencial. Una memoria secuencial simplemente trataría cada una de las experiencias como un elemento de una cola *First In First Out* (FIFO) , por lo que experiencias pasadas más antiguas irían siendo sustituidas por las más recientes. Así, las experiencias podrían ser escogidas aleatoriamente de forma uniforme para la fase de "*experience replay*".

El principal problema que tiene la utilización de memorias secuenciales es que es posible que el agente se vea muy sesgado por la elección de una serie de malas acciones de forma consecutiva y no sea capaz de aprender a realizar buenas acciones a partir de ese momento.

4.5.2. Memoria con prioridad

En 2015 Tom Schaul propuso una nueva forma de hacer el muestreo de dichas experiencias (como alternativa al muestreo uniforme de la memoria secuencial de la sección 4.5.1) [26].

Dicha propuesta consiste en dar un peso a cada una de las experiencias para dar a algunas prioridad sobre otras y eliminar el muestreo uniforme anterior. La idea detrás de esta alternativa es que hay algunas acciones que es posible que ocurran un menor número de veces pero que sean más significativas que el agente las aprenda. En el caso de muestrear de forma uniforme no sería posible dar a este tipo de experiencias una prioridad sobre las demás. La forma de asignar la probabilidad a cada una de las experiencias es, expresada matemáticamente como

$$P(i) = \frac{p_i^a}{\sum_{k \in I} p_k^a} \tag{4.4}$$

donde p_i es el valor de prioridad del elemento i calculado como la magnitud del error cometido más una constante distinta a cero para asegurar una prioridad no nula, a es un parámetro (con rango $[0, 1]$) que cuantifica la aleatoriedad de las elecciones y $\sum_{k \in I} p_k^a$ es un valor de normalización para expresar $P(i)$ en forma de probabilidad.

Hay que tener en cuenta que al poner prioridad a las experiencias, es posible que estemos introduciendo un sesgo a la hora de escoger con que experiencias entrenar. Por ello, controlaremos la actualización de pesos de la red para que sea inversamente proporcional al valor $P(i)$ anteriormente calculado.

4.6. "Learning from Demonstrations"

Como se ha explicado en la sección 4.4, el coste de la exploración inicial puede ser bastante grande y, en algunos casos, podría suponer unas pérdidas inasumibles por lo que harían inimplementables estos algoritmos en un entorno productivo. Adicionalmente, existían algunos entornos o problemas que el aprendizaje por refuerzo no podía superar, como es el ejemplo del videojuego *Montezuma's Revenge* o "La venganza de Montezuma".

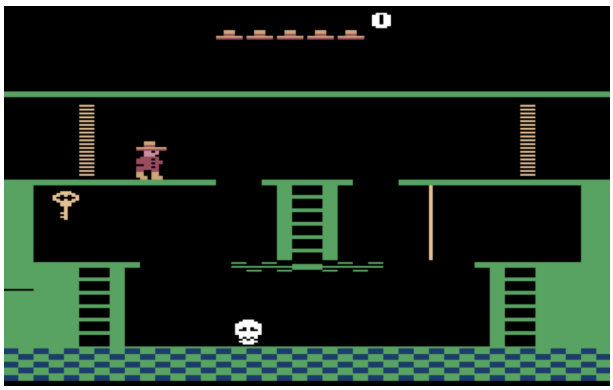


Figura 4.7: Primera sala del videojuego "La venganza de Montezuma". Imagen extraída del Blog de Arthur Juliani [27]

El videojuego "La venganza de Montezuma" es realmente especial y complicado de resolver por un agente debido a sus recompensas tan dispersas, lo que hace que el proceso de exploración se complique y el agente no llegue a converger.

Como solución al problema de la no asumible o infructuosa exploración inicial, *Google DeepMind* propone la alternativa de *Learning from Demonstrations* en su artículo "*Deep Q-Learning from Demonstrations*" [28]. Este avance supone una sinergia entre el aprendizaje supervisado (la parte de imitación sobre las experiencias fructíferas se propone como un problema de clasificación al uso) y el aprendizaje por refuerzo, siendo capaz de resolver problemas irresolubles anteriormente como en el caso del videojuego "La venganza de Montezuma" o de menguar los gastos de exploración inicial a la

hora de poner en producción estos algoritmos.

5

Accenture Smart Interactions

5.1. Descripción general del proyecto

ASI es un producto que oferta *Accenture* a portales web para la recomendación de anuncios de venta al por menor o *retail*. Este producto trata de recomendar los productos con mayor probabilidad de ser comprados por un usuario que interactúe en la página para así maximizar la afluencia de usuarios que navega del portal contratante del servicio hacia las que publicitan sus productos. De esta forma, el portal web contratante consigue que la eficiencia de la publicidad sea mayor, lo que le reportará mayores beneficios económicos.

Dentro de la arquitectura del producto ASI (figura 5.1), la elección de qué producto se mostrará a continuación recae sobre un sistema de recomendación como de los que se habla en el capítulo 2.4.3. Dicho sistema de recomendación funciona a base de un modelo de *Machine Learning* a cargo de *Accenture AI* que utiliza la información de las *cookies* de navegación del usuario así como interacciones anteriores del mismo para aumentar la probabilidad de dichas posibles navegaciones entre el portal contratante y la página que oferta los productos de venta al por menor.

5.2. Motivación y objetivos

5.2.1. Deep Learning

Hoy en día y gracias a las técnicas explicadas en el capítulo 3.4, se está pudiendo comprobar que los modelos de *Machine Learning* y en especial el *Deep Learning* basados en la acumulación de capas de neuronas para encontrar relaciones de mayor complejidad, tienen unos rendimientos realmente positivos en multitud de campos en general y en el de los sistemas de recomendación en particular [29].

Debido a este buen rendimiento, la unidad especializada en inteligencia artificial de *Accenture* (*Accenture AI*) está interesada en investigar el rendimiento de algoritmos de este estilo para valorar la

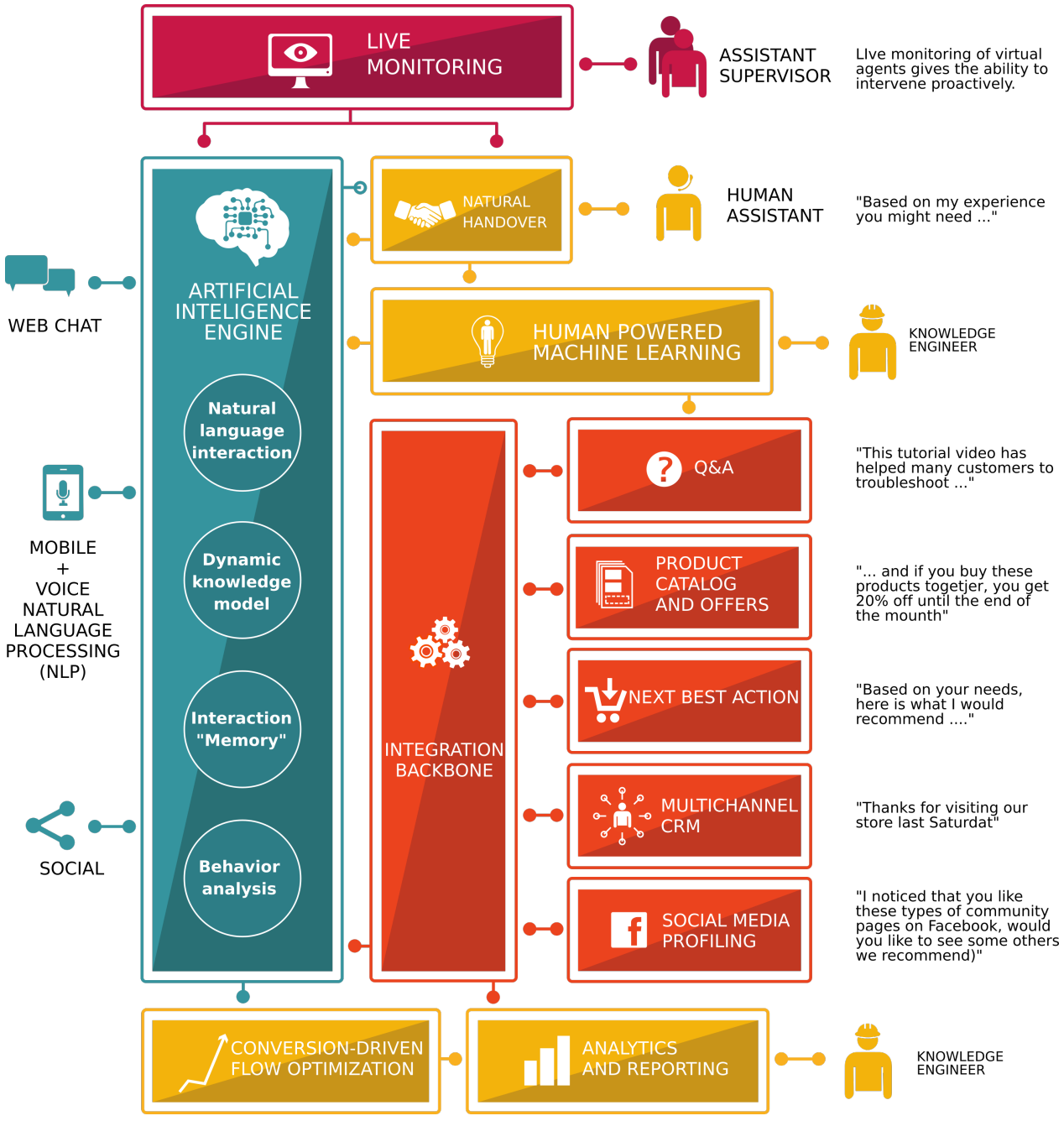


Figura 5.1: Explicación gráfica del producto Accenture Smart Interactions

implementación de estos nuevos modelos al producto ASI . Este interés se incrementa por la creciente facilidad de implementación de estas técnicas gracias a ciertas librerías que se explicarán más adelante.

5.2.2. Deep Reinforcement Learning

Utilizando el mejor de los algoritmos al que se pueda aspirar a través de las alternativas de los algoritmos tradicionales o modelos de *Machine Learning*/*Deep Learning* para los sistemas de recomendación se conseguiría la mejor recomendación con la mayor probabilidad de compra posible en la siguiente interacción con el usuario pero, ¿es esta recomendación la que mejores beneficios puede retornar a largo plazo?

Figura 5.2: Explicación práctica del efecto señuelo. Al introducir una nueva opción se puede controlar la preferencia hacia el producto deseado. Imagen adaptada del Blog de Bob Holmes: "The mind of an anthill" [30]

En el campo del marketing, existe un efecto psicológico denominado el "Efecto señuelo" o "Efecto de dominio asimétrico" [31]. Este fenómeno consiste en favorecer el cambio específico de preferencia de los consumidores entre un set de ítems a través de añadir otro ítem "asimétricamente dominante". Un ejemplo de este efecto puede ser la venta de palomitas de los cines, en los cuales se observó que añadiendo una tercera opción se conseguía aumentar las ventas de otra opcíon más rentable para el vendedor como en la figura 5.2.

Aquí es donde entra el campo del *Reinforcement Learning*. Como si de una partida de ajedrez se tratase, existen agentes basados en el paradigma del aprendizaje por refuerzo presentado en el capítulo 4.6 que intentan buscar recompensas a largo plazo a través de lo que se denomina movimientos subóptimos. Teniendo en cuenta la existencia del "efecto señuelo" anteriormente comentado, es comprensible que este tipo de estrategias puedan funcionar realmente bien en el contexto planteado.

Como podemos observar en la figura 5.3, este tipo de estrategias encajan a la perfección en el contexto de los sistemas de recomendación por varios motivos. El primero es que podemos ver al

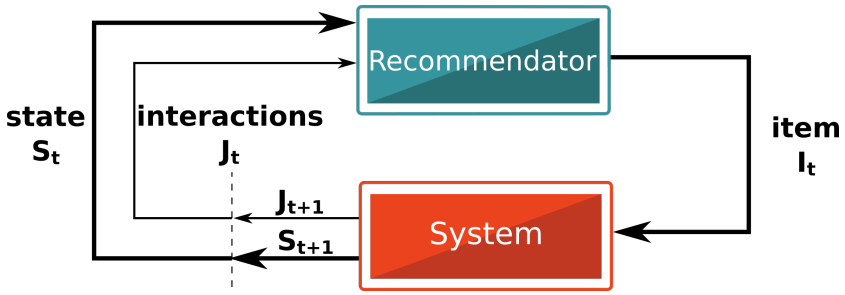


Figura 5.3: Diagrama de aprendizaje por refuerzo aplicado a la recomendación. En este caso las acciones de recomendación del agente serán los items; la recompensa será la interacción ante la visualización de dicho item por el usuarios; y el estado serán las variables a tener en cuenta por el recomendador.

agente como un "vendedor" que muestra los productos a un determinado usuario manteniendo el proceso iterativo tan característico de estos algoritmos, siendo:

- S: Conjunto de posibles estados de interacción y características del usuario a la hora de recomendar.
- It: Conjunto de ítems recomendables por el agente recomendador.
- In: Conjunto de interacciones posibles del usuario con el ítem.

El segundo motivo es que, como hemos comentado en la sección 4.4, estos agentes necesitan un proceso de exploración que, según cómo se elija, ayuda a aumentar métricas destinadas a cuantificar la variabilidad o novedad de las recomendaciones como pueden ser *Diversity* o *Novelty* y, por lo tanto, enriquecer la ganancia de información del usuario final, como se puede ver en los resultados de la tabla 6 del artículo "*DRN: A Deep Reinforcement Learning Framework for News Recommendation*" [32].

Por último, es fácil suponer que la visualización de un anuncio puede modificar la forma que tendrás de ver los siguientes, pudiendo estar más predispuesto a cierto tipo de anuncios o incluso a interactuar varias veces con elementos similares en un futuro. Prueba de ello son los buenos resultados obtenidos en los experimentos en vivo del algoritmo *REINFORCE* en *YouTube* [33].

Por ello, se ha visto que este tipo de agentes son realmente potentes a la hora de intentar resolver problemas utilizando como recompensa la simple interacción con el usuario que en cada caso se considere positiva, sin la necesidad de establecer una métrica de rendimiento al uso como las expuestas en la sección 2.4.2, cada una con su axioma/sesgo particular. En el blog de *OpenAI* (insertar-referencia-link) [34] se ha podido comprobar que las soluciones alcanzadas simplemente por la interacción resultan mucho más elegantes que en las que se especificaba una función de recompensa analítica a optimizar.

5.3. Experimentos

5.3.1. Entorno de ejecución

El desarrollo de las herramientas y *scripts* a utilizar en este estudio será a través de *Python*. Más concretamente se realizarán en la versión de *Python 3* y usando el formato *Notebook* que ofrece la herramienta *Jupyter*, ya que ofrece la posibilidad de exportar posteriormente el código y las salidas de cada ejecución en formato LaTeX. Las librerías utilizadas son:

Pandas Librería con la que podremos crear tablas indexadas para un manejo más sencillo y eficiente de la gran cantidad de datos de los que disponemos. Para más información tenemos la Documentación de la librería Pandas

Gym *Gym* [35] es una herramienta desarrollada por *OpenAI* destinada a proporcionar una serie de entornos contra los que poder ejecutar nuestros agentes de aprendizaje por refuerzo, estableciendo una interfaz unificada entre dichos entornos. Se puede encontrar más información en la página oficial de OpenAI-Gym.

Keras Librería de alto nivel para el diseño y ejecución de modelos predictivos basados en topologías de redes neuronales. Es capaz de ejecutar sobre *Tensorflow*, *CNTK* o *Theano*. En nuestro caso particular ejecutaremos nuestros modelos sobre *Tensorflow*.

Keras-RL Librería que proporciona la funcionalidad de los modelos de aprendizaje por refuerzo basados en *Q-Learning* que utilizaremos. Al tratarse de una librería con escasa documentación y aún en desarrollo, utilizaremos una de las ramas por aprobar, ya que ahí se encuentran los agentes *from Demonstrations* que vamos a utilizar. En concreto utilizaremos la versión que se encuentra en la rama de Jake Grigsby.

Matplotlib Librería necesaria para graficar los datos de los resultados obtenidos con el proceso para poder ver visualmente los resultados de una forma más explicativa y aclaratoria.

Numpy Librería de manejo de datos numéricos de forma vectorial y matricial.

5.3.2. Modelos predictivos de aprendizaje profundo

Objetivos y métricas de rendimiento

El objetivo de estos experimentos reside en explorar el rendimiento de distintos modelos de *Machine Learning* a la hora de recomendar ítems en un *dataset* de testeo en relación a un modelo basado en una recomendación aleatoria propuesto por el equipo de *Accenture Digital*, así como tiempos de ejecución y demás métricas de entrenamiento y validación.

Para comparar el rendimiento de los distintos modelos, *Accenture Digital* especificó una métrica de número de aciertos en el *target* a la hora de recomendar 10 ítems a cada uno de los usuarios, lo que en la literatura sería bastante parecido conceptualmente a un *Precision At k* donde $k = 10$ pero simplemente sumando los aciertos (por lo que denominaremos a esta métrica de rendimiento como

#Top10 a partir de ahora). Adicionalmente se han obtenido distintas métricas para cada uno de los modelos, tanto para la fase de entrenamiento como para la de test con el objetivo de poder observar anomalías si las hubiere y obtener una comparativa de tiempos de cara a la implementación de cada uno de los algoritmos en un entorno productivo.

El modelo aleatorio propuesto por *Accenture Digital* retorna de promedio un *#Top10* de 23. Este valor es un valor inherente al enunciado de la propuesta del reto, por lo que no será objetivo de este estudio de alternativas la verificación de dicha métrica.

Conjunto de datos

Los datos otorgados por *Accenture* corresponden a unos datos utilizados en un reto analítico propio de la unidad de *Accenture AI* para probar las estrategias de recomendación de distintos equipos sobre un conjunto de datos común. En la documentación del reto se dice que son datos sobre visualización y compra de películas de la web de un videoclub aunque no es comprobable porque los datos han sido otorgados de forma anonimizada.

Los datos constan de los siguientes archivos, descritos en un fichero *README.txt* de la siguiente forma:

COMPRAS.DAT Ítems adquiridos por los usuarios a lo largo del tiempo.

MAESTRO.DAT Características de los items.

MAESTROTARGET.DAT Items que son elegibles para el *target*.

README.TXT Descripción del contenido del resto de archivos.

TARGETTRAIN.DAT Usuarios con ítems adquiridos posteriormente.

USERTEST.DAT Usuarios pertenecientes al *target*.

VISITAS.DAT Ítems visitados por los usuarios a lo largo del tiempo.

Como podemos observar, en lo que a la fase de entrenamiento se refiere, tenemos dos archivos de interacciones (COMPRAS.DAT y VISITAS.DAT) de usuarios con ítems, generados con la herramienta que proporciona *Google Analytics*; y un archivo que contiene dos variables categóricas (una de 4 categorías y otra de 10) por ítem para los casos en los que queramos utilizar algoritmos que se basen en las características de dichos ítems. Los datos constan de:

Nº Items $7\,780$ items.

Nº Usuarios $235\,576$ usuarios.

Nº Interacciones $11\,815\,571$ interacciones (ya bien sean compras o visitas).

Densidad $11\,815\,571 / (7\,780 \times 235\,576) = 0{,}0064$. Es decir, una densidad del $0{,}64\,\%$.

Para generar los datos preparados necesarios para cada uno de los modelos que se utilizarán más adelante se ha preparado un *script* que tendrá varios modos de ejecución según las asunciones que

se harán de relevancia de los ítems o el tratamiento estático o temporal de los mismos. Cabe destacar que previamente se ha preparado un mapeo de cada uno de los identificadores de usuarios, ítems y atributos destinados a facilitar la ingesta de dichos identificadores por las capas de *Embeddings*, reservando el valor 0 para hacer el modelo robusto a ítems o atributos no presentes en el conjunto de entrenamiento. A continuación se pasará a explicar más en profundidad las especificaciones de cada uno de los modos de preparación de los datos.

Modo 1: Modelo estático y relevancia binaria

En este modo se utilizará una relevancia binaria, es decir, se supondrá que un ítem es relevante si la interacción del usuario con el ítem es de la categoría "compra" (valor numérico 1) y, por el contrario, se considerará que un ítem no resulta relevante para el usuario si la interacción pertenece a la categoría "visita" (valor numérico 0).

Al tratarse de un modelo que no supondrá un tratamiento temporal de la secuencia de interacciones de cada uno de los usuarios, no se tendrá en cuenta la columna que contiene el *timestamp* de cada una de las interacciones ni se hará un tratamiento especial en este aspecto.

Modo 2: Modelo estático y relevancia gradual

En esta ocasión, supondremos una relevancia gradual atendiendo a la proporción de eventos de "compra" con respecto a las interacciones totales de un usuario con un ítem. La formulación matemática de esta suposición de relevancia será

$$Rel_{u,i} = \frac{|Compras_{u,i}| + (\mu \cdot p)}{|Compras_{u,i}| + |Visitas_{u,i}| + \mu} \tag{5.1}$$

donde $|Compras_{u,i}|$ es el número de compras del ítem i por el usuario u, $|Visitas_{u,i}|$ es el número de visitas del usuario u al ítem i y los términos μ y p son necesarios para realizar un suavizado de Laplace debido a errores de la recopilación de los datos por parte de *Google Analytics*.

Al igual que en el caso del modo de ejecución anterior, no se hará ningún tratamiento especial con la columna que representa el instante de tiempo de cada una de las interacciones.

Modo 3: Modelo temporal y relevancia binaria

En este último modo, volveremos a suponer una relevancia binaria de la misma forma que lo hemos hecho en el modo que genera los datos para los modelos estáticos de relevancia binaria 5.3.2. Sin embargo, esta vez sí que utilizaremos la columna que contiene el instante de tiempo de la interacción para generar una lista de interacciones anteriores por cada una de las categorías. Es decir, para cada interacción añadiremos una lista con las "compras" y otra con las "visitas" anteriores a dicha interacción atendiendo a la columna del *timestamp* anteriormente nombrada.

Este modo no llegará a verse reflejado en ninguno de los modelos presentados debido a las limitaciones de computación de las que se disponía para el entrenamiento de un modelo que tenga en cuenta las anteriores características. El desarrollo del mismo servirá para futuros experimentos a cargo del equipo de *Accenture Digital*.

Modelos basados en contenido

Para estos modelos se utilizarán los datos generados a través del modo 1 (apartado 5.3.2) o del modo 2 (apartado 5.3.2) (según la asunción de relevancia que hagamos, binaria o gradual) atendiendo a las columnas de identificadores mapeados de usuario, atributo Nº 1 y atributo Nº 2.

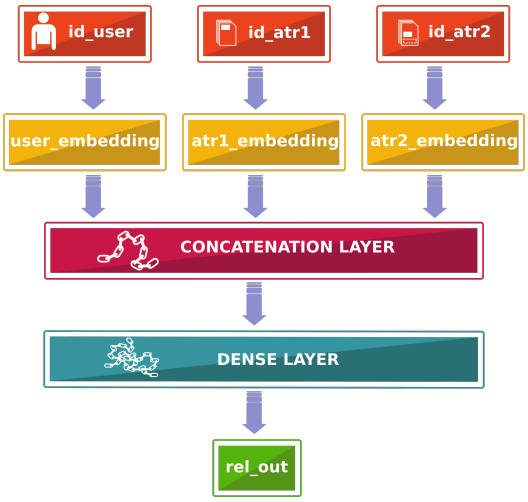


Figura 5.4: Arquitectura del modelo basado en contenido.

Como se puede ver en la figura 5.4, este modelo convertirá cada uno de los identificadores de usuario, atributo 1 y atributo 2 a un vector latente a través de la capa de *Embedding* explicada ya anteriormente en la sección 3.4. Posteriormente se concatenarán dichas capas para conectarla a una capa densa completamente conexa y posteriormente hacia la neurona de salida, que intentará predecir la relevancia del ítem en cuestión.

Modelos basados en filtrado colaborativo

En esta ocasión se volverán a utilizar los datos generados por los modos Nº1 y Nº2, pero esta vez utilizando únicamente los identificadores mapeados de usuario e ítem.

En la figura 5.5, estos modelos convertirán cada uno de los identificadores de usuario e ítem a un vector latente a través de la capa de *Embedding* y después se volverán a concatenar dichas capas para conectarla a una capa densa completamente conexa y posteriormente hacia la neurona de salida, que volverá a intentar predecir la relevancia del ítem en cuestión.

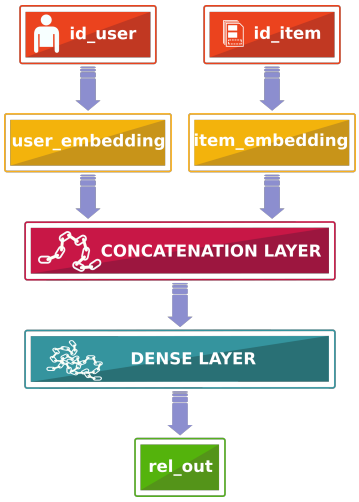


Figura 5.5: Arquitectura del modelo basado en filtrado colaborativo.

Modelos híbridos

En esta última alternativa, volveremos a utilizar los datos generados a través de los modos 1 y 2, de nuevo según la suposición de relevancia que queramos hacer en cada uno de los casos, pero utilizaremos todos los identificadores disponibles: los identificadores de usuario, ítem y los de ambos atributos.

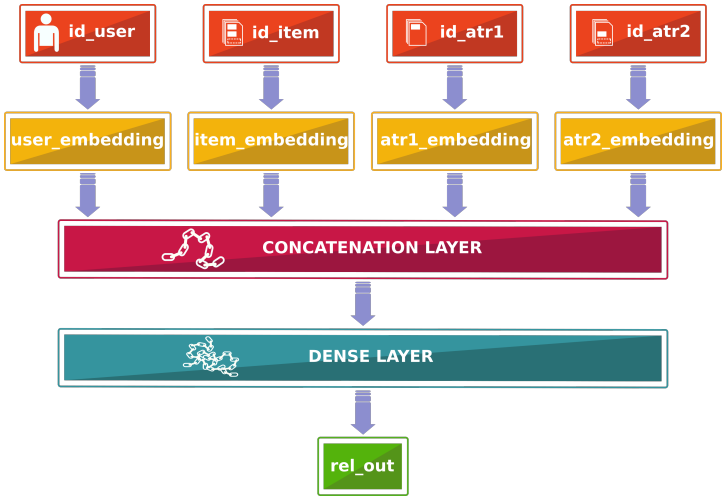


Figura 5.6: Arquitectura del modelo híbrido.

En la figura 5.6 vemos como en esta última alternativa, los modelos híbridos tendrán todos los identificadores como entrada conectados a su propia capa de *Embedding*, y continuaremos concatenando dichas capas para acabar con una capa densa completamente conexa con su correspondiente predicción.

Resultados totales

Ejecutando los modelos anteriormente expuestos para cada una de las suposiciones de relevancia expuestas, obtenemos la siguiente tabla comparativa:

Model	Mode	#Top10	%Top10	Accuracy	MSE	Time (s)
Random	Val & Test	23	0.423			
Static-Binary-EmbeddingRegressor-CB	Train			0.905	0.0828	278.3
	Val & Test	7	0.129	0.905	0.0830	43.2
Static-Binary-EmbeddingRegressor-CF	Train			0.905	0.0793	259.3
	Val & Test	51	0.938	0.905	0.0800	41.1
Static-Binary-EmbeddingRegressor-CFCB	Train			0.905	0.0785	297.8
	Val & Test	44	0.810	0.905	0.0793	45.7
Static-Weighted-EmbeddingRegressor-CB	Train			0.625	0.0162	358.4
	Val & Test	11	0.202	0.622	0.0206	45.9
Static-Weighted-EmbeddingRegressor-CF	Train			0.626	0.0142	344.9
	Val & Test	20	0.368	0.623	0.0188	39.6
Static-Weighted-EmbeddingRegressor-CFCB	Train			0.626	0.0140	387.3
	Val & Test	24	0.442	0.623	0.0188	48.8

Tabla 5.1: Tabla comparativa de resultados de modelos de deep learning. Puede observarse que solo se obtiene una mejora significativa de la métrica #Top10 en los modelos de relevancia binaria híbrido y el los basados exclusivamente en filtrado colaborativo, siendo este último el que mejor resultados retorna y menos tiempo tarda en entrenarse.

En ella podemos observar cómo la opción de modelo basado en filtrado colaborativo con una suposición de relevancia binaria (*Static-Binary-EmbeddingRegressor-CF*) obtiene la mayor puntuación en la métrica escogida (*#Top10*). Cabe destacar que no existe una gran discrepancia entre las métricas disponibles en la fase de entrenamiento con las métricas obtenidas en el conjunto de validación, lo que nos indica que no se está produciendo un *overfitting* (explicado en la sección 3.3) a los datos de entrenamiento.

También destaca que, pese a que ninguno de los modelos expuestos se exceden en el tiempo de entrenamiento, es precisamente el modelo ganador el que menor tiempo de entrenamiento necesita, por lo que podemos afirmar que la información de atributos que contienen los ítems no resulta de gran ayuda en este tipo de modelos predictivos y que , además, ralentizan el proceso de entrenamiento.

5.3.3. Modelos de aprendizaje por refuerzo

Una vez exploradas las alternativas anteriores, es de gran interés explorar el desempeño de distintos algoritmos de aprendizaje por refuerzo aplicados al problema de la publicidad *online* ya que es un contexto en el que existen movimientos que pueden retornar una recompensa a más largo plazo o efectos de similar carácter al “Efecto señuelo”.

Objetivos

En el presente experimento se analizarán los distintos rendimientos de los algoritmos DQN , *Dueling Deep Q-Networks* (DDQN) , *Double Deep Q-Networks* (DoubleDQN) y *Double Dueling Deep Q-Networks* (DoubleDDQN) contra un entorno de simulación debido al carácter interactivo necesario para este tipo de algoritmos.

Además, se explorarán las ventajas y desventajas de realizar un aprendizaje previo a la interacción con el entorno en las distintas versiones comentadas entariormente en las alternativa conocidas como *from demonstrations*

Entorno de simulación: RecoGym

El mayor problema a la hora de intentar comprobar el rendimiento de este tipo de algoritmos es que tienen que tener la capacidad de interactuar con un entorno. Si nos aferramos a los datos anteriores de "compras" y "visitas" nos es del todo imposible simular con exactitud todo el abanico de posibilidades de interacciones o secuencias de interacciones posibles. Afortunadamente, David Rohde y *Criteo AI Labs* idearon un entorno de simulación basado en la librería *OpenAI Gym* para exactamente dicho cometido en su artículo "RecoGym: A Reinforcement Learning Environment for the Problem of Product Recommendation in Online Advertising" [36].

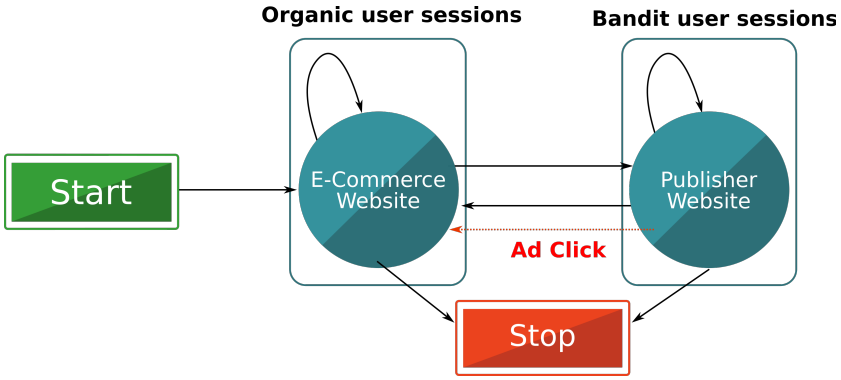


Figura 5.7: Diagrama de RecoGym. Imagen extraída del artículo "RecoGym: A Reinforcement Learning Environment for the Problem of Product Recommendation in Online Advertising" [36].

El entorno que utilizaremos define como recompensa para los agentes la suma de los *Click-through rate* de cada uno de los usuarios que conforman lo que se denomina un episodio. Un episodio se trata del periodo de ejecución definido por el entorno en el cual el agente puede obtener recompensas. Por ejemplo, en un videojuego, un episodio sería el conjunto de pasos que ha llevado al agente a "ganar" o a "perder".

RecoGym diferencia entre las *organic sessions* y las *bandit sessions* (figura 5.7). Las *organic sessions* serían simulaciones de interacciones no recomendadas por el agente (por lo que no podrán verse traducidas en una ganancia o pérdida de la recompensa) a partir de las cuales el agente podrá iniciar

su proceso de recomendación, por lo que entrará en las denominadas *bandit sessions*.

El entorno propuesto simula la interacción del usuario con cada ítem a través de una $Bernoulli(\sigma(P(u,i,t)))$ donde $\sigma(.)$ es una función logística sigmoidal y $P(u,i,t)$ es la probabilidad de que el usuario u vea orgánicamente el producto i en el momento t.

Por supuesto, se es consciente que el modelado de interacciones de este entorno no tiene por qué ajustarse del todo a una ejecución real ni a interacciones reales de usuarios en un entorno productivo. Sin embargo, resulta una aproximación bastante útil para hacer las comparativas que haremos en adelante. Además, al ser el entorno escogido un entorno con la funcionalidad propia de la librería *Gym*, ante una mejora o redefinición del mismo sería posible realizar la presente comparativa de alternativas sin necesidad de adaptar a nuestros agentes.

Agentes Deep Q-Learning

Utilizando la librería *keras-rl* especializada en crear agentes basados en redes neuronales (a través de la librería *keras*) de *reinforcement learning* para el lenguaje de programación *Python* será como testearemos cada una de las alternativas contra el entorno escogido.

Así,se entrenará cada uno de los modelos durante $1\ 000\ 000$ de pasos, utilizando el método de exploración ϵ *Greedy* con un valor inicial de $\epsilon = 1{,}0$ (con lo que comenzará siempre con exploraciones aleatorias) que irá decayendo de forma lineal hasta un mínimo de $\epsilon = 0{,}1$. Se llegará hasta dicho valor cuando lleve un $80\,\%$ de los pasos totales de entrenamiento. Para la fase de test se mantendrá un valor de $\epsilon = 0{,}05$, ya que es recomendable continuar explorando siempre. Por otro lado, se utilizará un factor de decadencia $\gamma = 0{,}999$ de la función Q que se intenta estimar a través de cada uno de los estimadores pertinentes en cada caso.

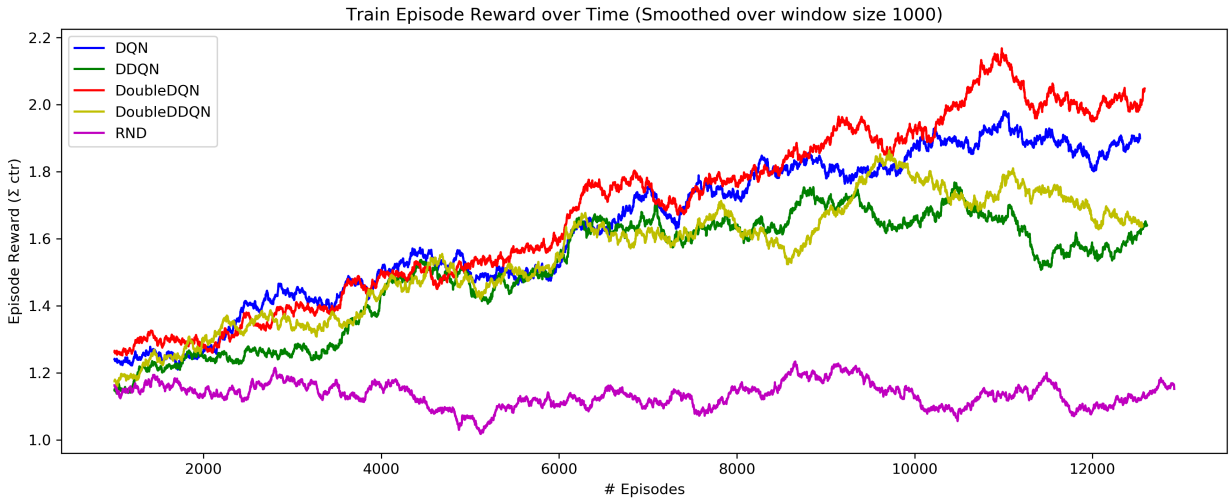


Figura 5.8: Comparación entre los entrenamientos de los agentes con distintas modificaciones de Deep Q-Networks.

En la figura 5.8 podemos observar el rendimiento de cada uno de los agentes a lo largo de los episodios durante la fase de entrenamiento. Los datos brutos obtenidos han de ser suavizados de acuerdo a una ventana debido a lo dispares que resultan unos resultados sobre otros ya que lo que nos interesa visualizar es la tendencia del algoritmo. En el gráfico podemos ver la comparativa de los algoritmos estudiados contra un agente puramente aleatorio (*Random*). Así, podemos ver que la alternativa de DoubleDQN parece ser la que mejor aprende a través de los episodios.

Por otro lado, en la figura 5.9 tenemos el rendimiento de esos mismos agentes en la fase de test, es decir, símplemente con el valor ϵ menguado en gran medida, pero sin ser nulo. En esta ocasión podemos ver cómo resulta ser el algoritmo de DoubleDDQN el que mejor rendimiento tiene pero con un coeficiente de variación (*CV*) ligeramente superior, por lo que dicha diferencia se puede dar simplemente por la aleatoriedad que introduce ese valor ϵ. De cualquier forma, vemos como son las alternativas *Double* las que mejor rendimiento consiguen en la fase de test, reafirmando los resultados de *Hado van Hasselt* en su artículo "Deep Reinforcement Learning with Double Q-learning" [23].

Agentes Deep Q-Learning from Demonstrations

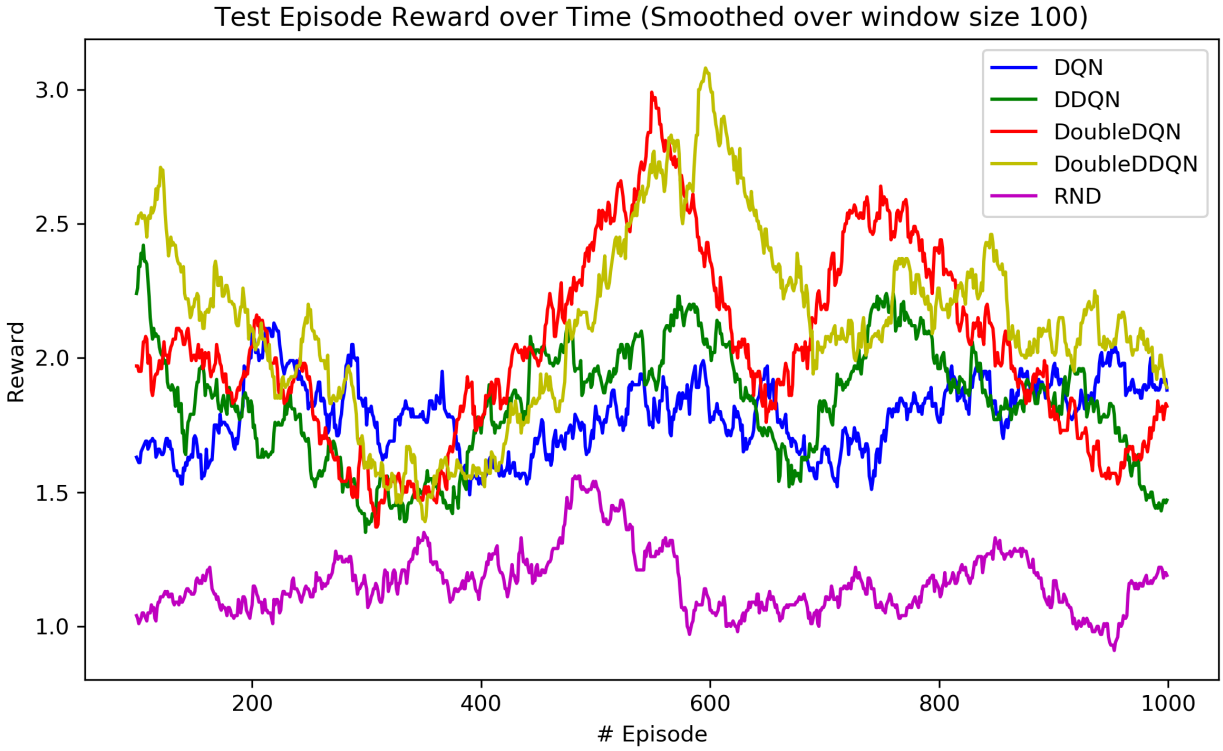
Los agentes *from Demonstrations* necesitan de unas condiciones especiales para su ejecución. Necesitan de un lote de experiencias previo al que atenerse para realizar una fase de entrenamiento *offline* anterior. También, es necesario que tengan una memoria con prioridad (subsección 4.5.2) en lugar de una secuencial.

En la figura 5.10 se puede ver como la mayoría de los agentes *from Demonstrations* siguen un proceso de aprendizaje en el que van aprendiendo a realizar recomendaciones a medida que avanzan los episodios. Sin embargo se puede ver que, curiosamente, la opción Dueling Deep Q-Learning from Demonstrations (DDQNfD) no consigue aprender con la interacción con el entorno. Es más, no sólo no mejora su rendimiento sino que lo empeora cada vez más, llegando a tener un rendimiento peor que la propia recomendación aleatoria (*Random*).

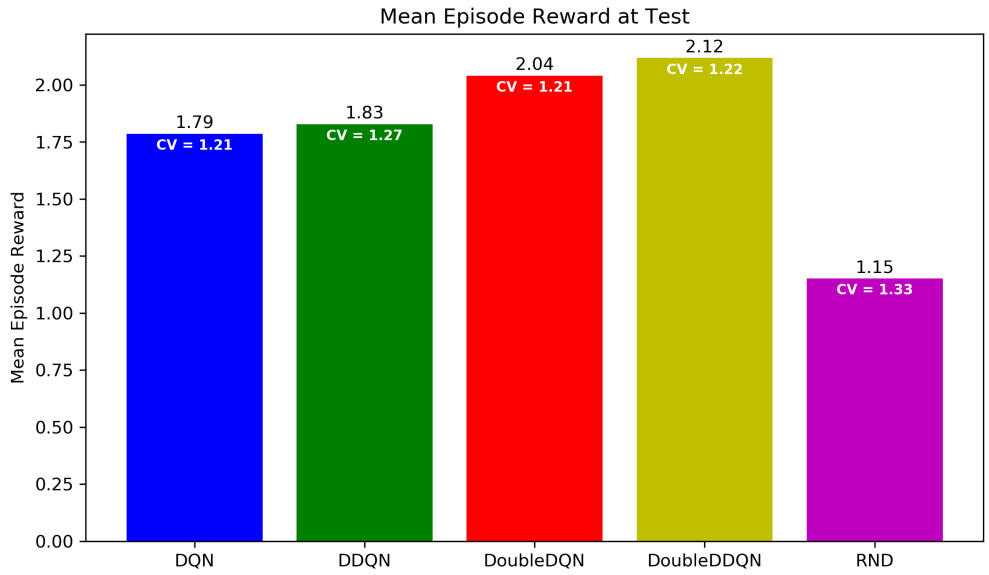
Como ya se podía comprobar en la fase de entrenamiento de los agentes, en la figura 5.11 vemos que la versión DDQNfD tiene un rendimiento inferior al agente puramente aleatorio, por lo que se concluye que no sería una buena alternativa a implementar en producción.

También se puede observar cómo el rendimiento final no suele superar a las versiones que no han tenido ese aprendizaje *offline* del que se beneficiaban este tipo de agentes. Es más, podemos ver cómo en algunos casos resulta ser la opción al uso la que mejora el rendimiento a su versión *from Demonstrations*. Este efecto se puede deber a que el agente se ve muy sesgado a la colección de experiencias previas recopiladas, estableciendo una especie de cota superior al rendimiento alcanzable por el agente.

Por último, podemos ver en la figura 5.12 cómo las versiones *from Demonstrations* mejoran el



(a) Recompensas durante los episodios de test.



(b) Recompensas medias durante el test.

Figura 5.9: Comparación entre los tests de los agentes con distintas modificaciones de Deep Q-Networks.

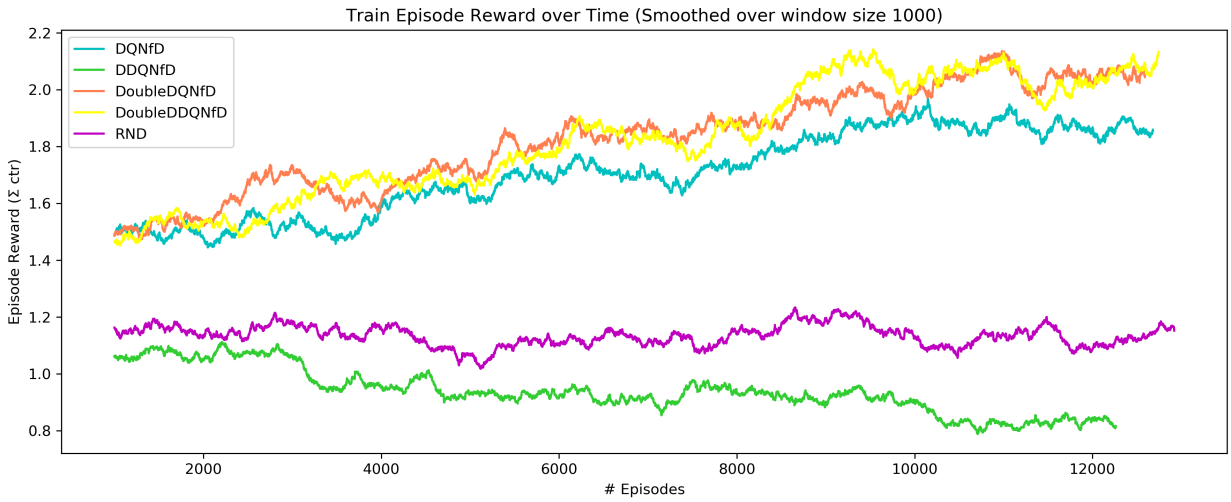
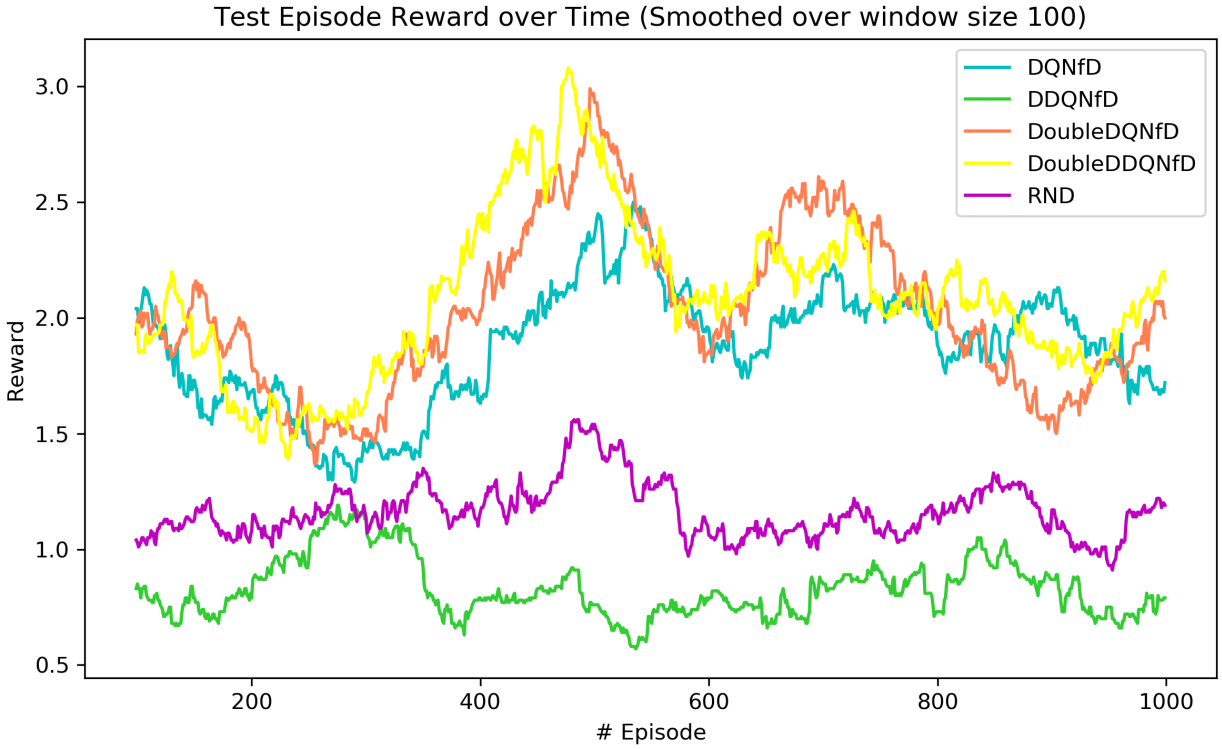
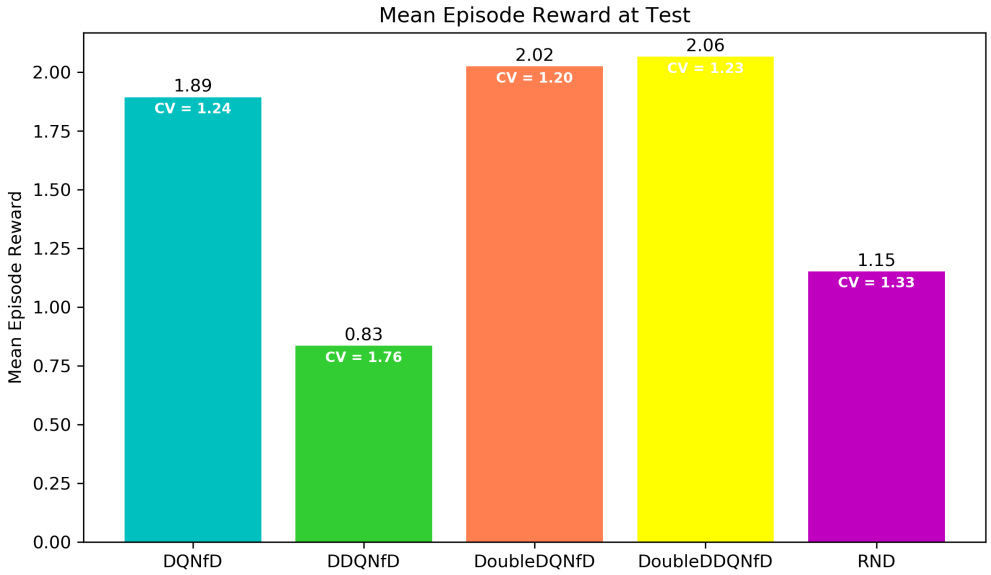


Figura 5.10: Comparación entre los entrenamientos de los agentes con distintas modificaciones de Deep Q-Networks en su versión from Demonstrations.

rendimiento en los primeros episodios de la interacción con el entorno (a excepción de DDQNfD). Esta ventaja puede ser aprovechada a la hora de poner en un entorno productivo alguna alternativa de estos agentes y no comenzar con los costes tan grandes típicos de la exploración inicial.



(a) Recompensas durante los episodios de test.



(b) Recompensas medias durante el test.

Figura 5.11: Comparación entre los tests de los agentes con distintas modificaciones de Deep Q-Networks en su versión from Demonstrations.

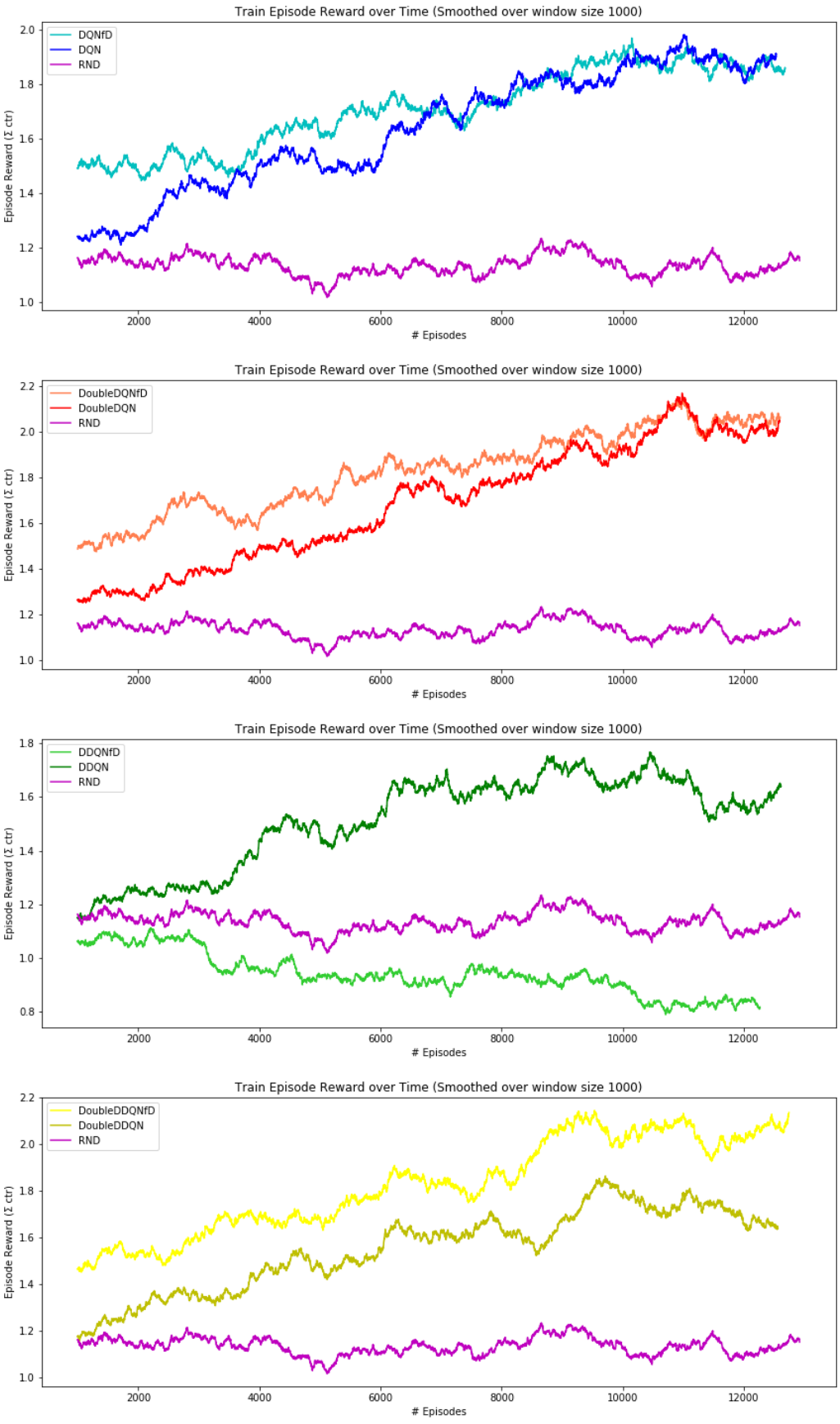


Figura 5.12: Comparativas de entrenamiento de cada una de las alternativas con su versión from Demonstrations.

6

Conclusiones y trabajo futuro

6.1. Conclusiones

A la vista de los resultados obtenidos en el presente *Trabajo de Fin de Máster* sobre el de ciertos modelos de *machine learning*/*deep learning* aplicados a un sistema de recomendación podemos comprobar como, con los datos de los que se dispone:

- La mejor opción resulta ser la opción que utiliza el filtrado colaborativo tanto en rendimiento como en tiempo de entrenamiento.
- A pesar de tener datos sobre los atributos de los ítems a recomendar, esta información sólo introduciría ruido en el modelo y haría más costoso el entrenamiento del mismo al tratarse de apenas dos atributos con 4 y 10 categorías cada uno, por lo que se entiende que resultaban poco explicativos.
- La suposición de relevancia de forma binaria retorna mejores rendimientos que la suposición de relevancia gradual.

Adicionalmente, se ha podido comprobar que en los algoritmos propuestos de *reinforcement learning*:

- El algoritmo que mejores resultados retorna es la opción de DoubleDDQN , que combina todas las mejoras aplicables a DQN vistas en el capítulo 4.6.
- Las opciones con un aprendizaje *offline* previo (alternativas *from Demonstrations*) mejoran los rendimientos en los primeros episodios, minimizando las posibles pérdidas iniciales de implementación en un entorno productivo.
- El rendimiento final de los agentes de aprendizaje por refuerzo *from Demonstrations* alcanzan un rendimiento inferior a los agentes que no tienen experiencias previas, por lo que parece que las experiencias anteriores sesgan al agente estableciendo una cota superior al posible rendimiento del mismo.
- La opción del agente DDQNfD (DDQN en su versión *from Demonstratons*) resulta que obtiene un rendimiento notablemente inferior incluso al agente completamente aleatorio (*Random*), por lo que no resultaría una buena opción a la hora de solucionar el problema que nos incumbe.

6.2. Trabajo futuro

En lo referente a los agentes de *machine learning*/*deep learning* al uso, sería de gran interés destinar más poder de computación a probar modelos temporales basados en Long Short-Term Memories (LSTM's) o en Convolutional Neural Networks (CNN's) de una dimensión, que son modelos que retornan bastante buenos resultados en el análisis de información temporal. Para ello, se podrán utilizar los datos de salida del modo preparado para modelos temporales de relevancia binaria (5.3.2) de tratamiento de los datos presentado en el presente *Trabajo de Fin de Máster*.

Por otro lado, el trabajo futuro interesante para los agentes recomendadores basados en *reinforcement learning* puede desarrollarse en multitud de ámbitos. Estos pueden ser desde repensar la forma de otorgar la recompensa hasta estudiar el algoritmo de exploración óptimo (como UCB), pasando por probar otros tipos de agentes.

En el caso de optar por mejorar la asignación de la recompensa a las acciones, se podría idear algún tipo de *ranking* de acciones realizadas (lo que significa un *ranking* de ítems) y se asumirá que todas las acciones del top escogido se realizan para promover visualizaciones simultáneas de anuncios que mejoren las métricas. En otras palabras, esto haría que el agente fuese capaz de averiguar qué anuncios funcionan bien juntos, pero de manera simultánea.

Por último, sería muy interesante ampliar el estudio del rendimiento de otro tipo de agentes en el problema presentado. Estos nuevos agentes podrían basarse en métodos como *Actor-Critic* o similares (Advantage Actor-Critic (A2C) , Asynchronous Advantage Actor-Critic (A3C) , etc.) en lugar del *Deep Q-Learning* aquí presentado.

Bibliografía

[1] L. Boratto, S. Carta, G. Fenu, and R. Saia, "Semantics-aware content-based recommender systems: Design and architecture guidelines," *Neurocomputing*, vol. 254, pp. 79–85, sep 2017.

[2] S. Zhang, L. Yao, Y. I. Tay, A. Sun, and Y. Tay, "Deep Learning based Recommender System: A Survey and New Perspectives," *ACM Computing Surveys*, vol. 1, no. 7, p. 35, 2018.

[3] P. Castells, S. Vargas, and J. Wang, "Novelty and Diversity Metrics for Recommender Systems: Choice, Discovery and Relevance," tech. rep., 2011.

[4] L. Sherman and J. Deighton, "Banner advertising: Measuring effectiveness and optimizing placement," *Journal of Interactive Marketing*, vol. 15, pp. 60–64, jan 2001.

[5] TrustRadius, "The Complete Guide to A/B Testing and False Positives," 2018.

[6] R. Sabbatini, "Neurons and synapses: the history of its discovery." http://www.cerebromente.org.br/n17/history/neurons3_i.htm, 2003.

[7] Wikipedia, "Neurona," 2019.

[8] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[9] F. Sancho Caparrini, "Redes Neuronales: una visión superficial." http://www.cs.us.es/~fsancho/?e=72, 2018.

[10] D. E. Ruineihart, G. E. Hint, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., UCSD, 1985.

[11] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *ICLR*, p. 15, 2015.

[12] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," in *Proceedings of tMachne Learning Research*, pp. 315–323, jun 2011.

[13] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, vol. 9, pp. 249–256, 2006.

[14] D. Stutz, "Understanding the difficulty of training deep feedforward neural networks explanation." http://davidstutz.de/understanding-the-difficulty-of-training-deep-feedfoward-neural-networks/, 2017.

[15] Y. Yue, "What is high bias and high variance in machine learning terminology in simplest terms?." https://www.quora.com/What-is-high-bias-and-high-variance-in-machine-learning-terminology-in-simplest-terms, 2018.

[16] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[17] Jason Brownlee, "Dropout Regularization in Deep Learning Models With Keras," 2016.

[18] O. Levy and Y. Goldberg, “Neural Word Embedding as Implicit Matrix Factorization,” tech. rep., 2015.

[19] Psicoactiva, “Skinner y en Condicionamiento Operante,” 2019.

[20] Shweta Bhatt, “5 Things You Need to Know about Reinforcement Learning,” 2018.

[21] M. N. Katehakis and A. F. Veinott, “The Multi-Armed Bandit Problem: Decomposition and Computation,” *Mathematics of Operations Research*, vol. 12, pp. 262–268, may 1987.

[22] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, “Resource Management with Deep Reinforcement Learning,” tech. rep., 2016.

[23] H. van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-learning,” *Association for the Advancement of Artificial Intelligence*, p. 13, sep 2015.

[24] Z. Wang, T. Schaul, M. Hessel, and M. Lanctot, “Dueling Network Architectures for Deep Reinforcement Learning Hado van Hasselt,” tech. rep., 2016.

[25] Y. Wang, K. Velswamy, and B. Huang, “A Long-Short Term Memory Recurrent Neural Network Based Reinforcement Learning Controller for Office Heating Ventilation and Air Conditioning Systems,” *Processes*, vol. 5, p. 46, aug 2017.

[26] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized Experience Replay,” in *ICLR*, p. 21, nov 2016.

[27] A. I. Medium, “On “solving” Montezuma’s Revenge - Arthur Juliani - Medium,” 2018.

[28] T. Hester, G. Deepmind, O. Pietquin, M. Lanctot, T. Schaul, D. Horgan, J. Quan, A. Sendonaris, G. Dulac-Arnold, J. Agapiou, and J. Z. Leibo, “Deep Q-learning from Demonstrations,” tech. rep., 2017.

[29] H. Wang, N. Wang, and D. Y. Yeung, “Collaborative deep learning for recommender systems,” *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, vol. 2015-Augus, pp. 1235–1244, 2015.

[30] B. Holmes, “The mind of an anthill,” sep 2018.

[31] J. Huber, J. W. Payne, and C. Puto, “Adding Asymmetrically Dominated Alternatives: Violations of Regularity and the Similarity Hypothesis,” *Journal of Consumer Research*, vol. 9, p. 90, jun 1982.

[32] G. Zheng, F. Zhang, Z. Zheng, Y. Xiang, N. J. Yuan, X. Xie, and Z. Li, “DRN: A Deep Reinforcement Learning Framework for News Recommendation,” *International World Web Conference Comittee*, p. 10, 2018.

[33] M. Chen, A. Beutel, P. Covington, S. Jain, F. Belletti, and E. H. Chi, “Top-K Off-Policy Correction for a REINFORCE Recommender System,” in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining - WSDM ’19*, (New York, New York, USA), pp. 456–464, ACM Press, 2019.

[34] P. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei, “Deep reinforcement learning from human preferences,” tech. rep., jun 2017.

[35] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “OpenAI Gym,” tech. rep., jun 2016.

[36] D. Rohde, S. Bonner, T. Dunlop, F. Vasile, and A. Karatzoglou, “RecoGym: A Reinforcement Learning Environment for the Problem of Product Recommendation in Online Advertising,” tech. rep., 2018.

Acrónimos

A2C Advantage Actor-Critic.

A3C Asynchronous Advantage Actor-Critic.

AdaGrad *Adaptative Gradient Algorithm*.

ADAM *ADAptative Moment estimation*.

ASI *Accenture Smart Interactions*.

CNN's Convolutional Neural Networks.

CNNs Convolutional Neural Nets.

CTR Click-through rate.

DDQN *Dueling Deep Q-Networks*.

DDQNfD Dueling Deep Q-Learning from Demonstrations.

DoubleDDQN *Double Dueling Deep Q-Networks*.

DoubleDQN *Double Deep Q-Networks*.

DQN *Deep Q Network*.

FIFO *First In First Out*.

kNN k Nearest Neighbours.

LSTM's Long Short-Term Memories.

MAE Mean Absolute Error.

MDPs *Markov Decission Processes*.

MSE Mean Squared Error.

nDCG Normalized Discounted Cumulative Gain.

ReLU Rectified Linear Unit.

RMSE Root Mean Squared Error.

RMSProp *Root Mean Square Propagation*.

RNNs Recursive Neural Nets.

SVD Singular Value Decomposition.

UCB *Upper Confidence Bound*.

Apéndices

A

Comparativa de rendimientos entre DQN y Double DQN

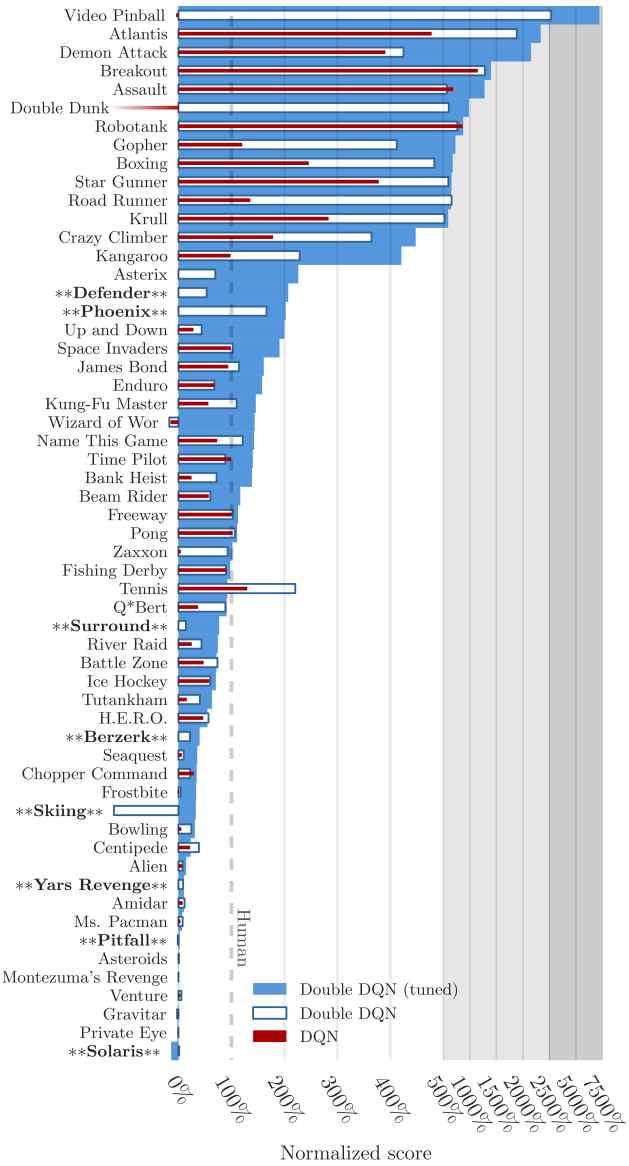


Figura A.1: Comparativa de rendimiento entre DQN y Double DQN. Imagen estraída del artículo "Deep Reinforcement Learning with Double Q-learning" [**?**]

B

Ejemplo de modelo predictivo de relevancia binaria

B.1. Imports

```
In [1]: from keras.layers import Input, Embedding, Dot, Reshape
        from keras.layers import Dense, Flatten, concatenate, Dropout, BatchNormalization
        from keras.models import Model
        from keras.callbacks import ModelCheckpoint
        from keras.optimizers import Adam
        from keras.utils import plot_model
        import pandas as pd
        import numpy as np
        import random
        import time
        from itertools import product

        import os,sys,inspect
        currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.currentframe())))
        parentdir = os.path.dirname(currentdir)
        sys.path.insert(0,parentdir)

        from OwnCodeAndMetrics import *

        %matplotlib inline
        np.random.seed(123)
```

```
Using TensorFlow backend.
```

B.2. Function BuildModel

```
In [2]: def buildModel(df_user_item):

            num_users = len(df_user_item["user"].unique())+1
            num_items = len(df_user_item["item"].unique())+1
            latent_factors_mf = 4
            embedding_cf_size = 10
            atrib_embedding_size = 4

            # All inputs are 1-dimensional
```

```
user = Input(name = 'user', shape = [1])
item = Input(name = 'item', shape = [1])
atr1 = Input(name = 'atr1', shape = [1])
atr2 = Input(name = 'atr2', shape = [1])


# Embedding the user (shape will be (None, 1, embedding_size))
user_embedding = Embedding(name = 'user_embedding_to_dot',
                           input_dim = len(df_user_item["user"].unique())+1,
                           output_dim = embedding_cf_size)(user)




# Embedding the item (shape will be (None, 1, embedding_size))
item_embedding = Embedding(name = 'item_embedding_to_dot',
                           input_dim = len(df_user_item["item"].unique())+1,
                           output_dim = embedding_cf_size)(item)

# Embedding the item atribute 1 (shape will be (None, 1, embedding_size))
atr1_embedding = Embedding(name = 'atr1_embedding',
                           input_dim = len(df_user_item["atributo1"].unique())+1,
                           output_dim = atrib_embedding_size)(atr1)



# Embedding the item atribute 2 (shape will be (None, 1, embedding_size))
atr2_embedding = Embedding(name = 'item_atr2_embedding',
                           input_dim = len(df_user_item["atributo2"].unique())+1,
                           output_dim = atrib_embedding_size)(atr2)

user_flat = Flatten()(user_embedding)
item_flat = Flatten()(item_embedding)
atr1_flat = Flatten()(atr1_embedding)
atr2_flat = Flatten()(atr2_embedding)

# Concatenate the Embedding layers
conc = concatenate([(user_flat),
                    (item_flat),
                    (atr1_flat),
                    (atr2_flat)])


d1 =    Dense(64, activation='relu', name = "Dense1")(conc)

out =   Dense(1, activation = 'sigmoid')(d1)

model = Model(inputs = [user, item, atr1, atr2], outputs = out)
# Compile using specified optimizer and loss
model.compile(optimizer = Adam(lr=0.0005), loss = 'mse', metrics = ["acc"])

return model
```

B.2.1. Loading Data

```
In [3]: train = pd.read_pickle("../../ProcessedData/(Total)Static-UserItem-MatrixBinary")
        items = pd.read_pickle("../../ProcessedData/ItemContent")

        train = train.merge(items,how='left', left_on = "item", right_on= "item")
        train = train.sample(frac=1).reset_index(drop=True)
```

B.3. Train

```
In [4]: #gen = generate_batch(train, batch_size)

        batch_size = 1000000
        path_model = "./Trained/Static-Binary-EmbeddingRegressor-CFCB"
        mcp = ModelCheckpoint(path_model, monitor="val_loss",
                              save_best_only=True, save_weights_only=False)

        # Train
        model = buildModel(train)
        print(model.summary())

        start_time = time.time()
        history_train = model.fit([train["user"], train["item"],
                                   train["atributo1"], train["atributo2"]]
                                    , train["rel"]
                                    , batch_size=batch_size, epochs=30
                                    , validation_split=0.1
                                    , callbacks=[mcp]
                                    , verbose = 1)
        delta_time_train = time.time() - start_time

        dic = {
            "Model": "Static-Binary-EmbeddingRegressor-CFCB",
            "Mode": "Train",
            "Time": delta_time_train,
            "MSE": min(history_train.history["loss"]),
            "Accuracy": max(history_train.history["acc"])
        }
        df_results = save_results(dic)
        df_results
```

```
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
user (InputLayer)            (None, 1)                 0
_________________________________________________________________________________________________
item (InputLayer)            (None, 1)                 0
_________________________________________________________________________________________________
atr1 (InputLayer)            (None, 1)                 0
_________________________________________________________________________________________________
atr2 (InputLayer)            (None, 1)                 0
_________________________________________________________________________________________________
user_embedding_to_dot (Embeddin (None, 1, 10)          2355770     user[0][0]
```

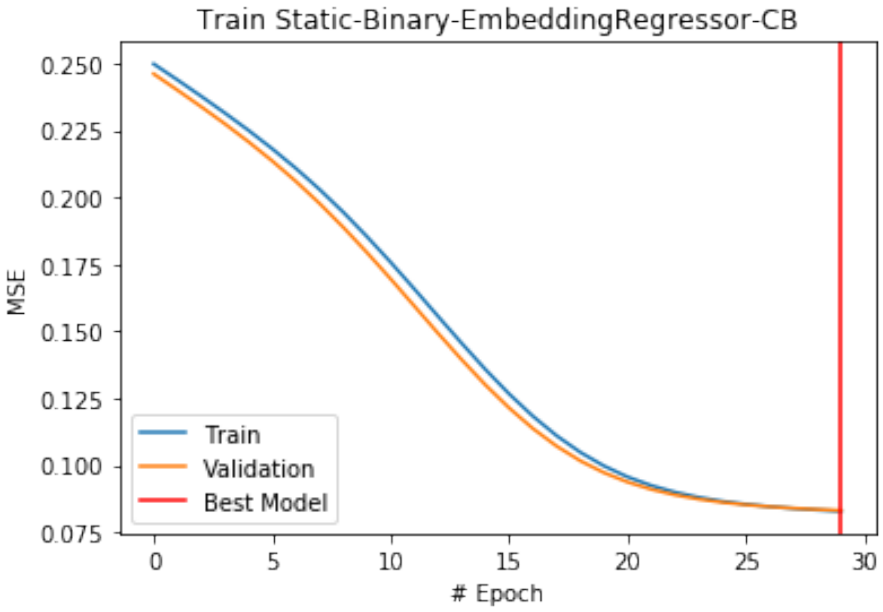
```
____________________________________________________________________________________________________
item_embedding_to_dot (Embeddin (None, 1, 10)        77810       item[0][0]
____________________________________________________________________________________________________
atr1_embedding (Embedding)      (None, 1, 4)         16          atr1[0][0]
____________________________________________________________________________________________________
item_atr2_embedding (Embedding) (None, 1, 4)         84          atr2[0][0]
____________________________________________________________________________________________________
flatten_1 (Flatten)             (None, 10)           0           user_embedding_to_dot[0][0]
____________________________________________________________________________________________________
flatten_2 (Flatten)             (None, 10)           0           item_embedding_to_dot[0][0]
____________________________________________________________________________________________________
flatten_3 (Flatten)             (None, 4)            0           atr1_embedding[0][0]
____________________________________________________________________________________________________
flatten_4 (Flatten)             (None, 4)            0           item_atr2_embedding[0][0]
____________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 28)           0           flatten_1[0][0]
                                                                 flatten_2[0][0]
                                                                 flatten_3[0][0]
                                                                 flatten_4[0][0]
____________________________________________________________________________________________________
Dense1 (Dense)                  (None, 64)           1856        concatenate_1[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                 (None, 1)            65          Dense1[0][0]
====================================================================================================
Total params: 2,435,601
Trainable params: 2,435,601
Non-trainable params: 0
____________________________________________________________________________________________________
None
Train on 5394743 samples, validate on 599416 samples
Epoch 1/30
5394743/5394743 - 10s 2us/step - loss: 0.2448 - acc: 0.8144 - val_loss: 0.2412 - val_acc: 0.8942
Epoch 2/30
5394743/5394743 - 10s 2us/step - loss: 0.2389 - acc: 0.9013 - val_loss: 0.2350 - val_acc: 0.9045
Epoch 3/30
5394743/5394743 - 10s 2us/step - loss: 0.2325 - acc: 0.9048 - val_loss: 0.2284 - val_acc: 0.9045
Epoch 4/30
5394743/5394743 - 10s 2us/step - loss: 0.2258 - acc: 0.9048 - val_loss: 0.2213 - val_acc: 0.9045
Epoch 5/30
5394743/5394743 - 10s 2us/step - loss: 0.2185 - acc: 0.9048 - val_loss: 0.2138 - val_acc: 0.9045
Epoch 6/30
5394743/5394743 - 10s 2us/step - loss: 0.2107 - acc: 0.9048 - val_loss: 0.2056 - val_acc: 0.9045
Epoch 7/30
5394743/5394743 - 10s 2us/step - loss: 0.2024 - acc: 0.9048 - val_loss: 0.1969 - val_acc: 0.9045
Epoch 8/30
5394743/5394743 - 10s 2us/step - loss: 0.1934 - acc: 0.9048 - val_loss: 0.1876 - val_acc: 0.9045
Epoch 9/30
5394743/5394743 - 9s 2us/step - loss: 0.1838 - acc: 0.9048 - val_loss: 0.1777 - val_acc: 0.9045
Epoch 10/30
5394743/5394743 - 9s 2us/step - loss: 0.1738 - acc: 0.9048 - val_loss: 0.1675 - val_acc: 0.9045
Epoch 11/30
5394743/5394743 - 9s 2us/step - loss: 0.1635 - acc: 0.9048 - val_loss: 0.1571 - val_acc: 0.9045
Epoch 12/30
5394743/5394743 - 9s 2us/step - loss: 0.1530 - acc: 0.9048 - val_loss: 0.1467 - val_acc: 0.9045
Epoch 13/30
5394743/5394743 - 9s 2us/step - loss: 0.1427 - acc: 0.9048 - val_loss: 0.1367 - val_acc: 0.9045
Epoch 14/30
```

```
5394743/5394743 - 9s 2us/step - loss: 0.1328 - acc: 0.9048 - val_loss: 0.1272 - val_acc: 0.9045
Epoch 15/30
5394743/5394743 - 9s 2us/step - loss: 0.1236 - acc: 0.9048 - val_loss: 0.1186 - val_acc: 0.9045
Epoch 16/30
5394743/5394743 - 9s 2us/step - loss: 0.1153 - acc: 0.9048 - val_loss: 0.1110 - val_acc: 0.9045
Epoch 17/30
5394743/5394743 - 10s 2us/step - loss: 0.1081 - acc: 0.9048 - val_loss: 0.1045 - val_acc: 0.9045
Epoch 18/30
5394743/5394743 - 9s 2us/step - loss: 0.1019 - acc: 0.9048 - val_loss: 0.0990 - val_acc: 0.9045
Epoch 19/30
5394743/5394743 - 9s 2us/step - loss: 0.0968 - acc: 0.9048 - val_loss: 0.0946 - val_acc: 0.9045
Epoch 20/30
5394743/5394743 - 9s 2us/step - loss: 0.0927 - acc: 0.9048 - val_loss: 0.0911 - val_acc: 0.9045
Epoch 21/30
5394743/5394743 - 11s 2us/step - loss: 0.0895 - acc: 0.9048 - val_loss: 0.0884 - val_acc: 0.9045
Epoch 22/30
5394743/5394743 - 11s 2us/step - loss: 0.0869 - acc: 0.9048 - val_loss: 0.0863 - val_acc: 0.9045
Epoch 23/30
5394743/5394743 - 11s 2us/step - loss: 0.0849 - acc: 0.9048 - val_loss: 0.0846 - val_acc: 0.9045
Epoch 24/30
5394743/5394743 - 10s 2us/step - loss: 0.0834 - acc: 0.9048 - val_loss: 0.0833 - val_acc: 0.9045
Epoch 25/30
5394743/5394743 - 10s 2us/step - loss: 0.0821 - acc: 0.9048 - val_loss: 0.0823 - val_acc: 0.9045
Epoch 26/30
5394743/5394743 - 9s 2us/step - loss: 0.0811 - acc: 0.9048 - val_loss: 0.0815 - val_acc: 0.9045
Epoch 27/30
5394743/5394743 - 10s 2us/step - loss: 0.0803 - acc: 0.9048 - val_loss: 0.0808 - val_acc: 0.9045
Epoch 28/30
5394743/5394743 - 10s 2us/step - loss: 0.0796 - acc: 0.9048 - val_loss: 0.0802 - val_acc: 0.9045
Epoch 29/30
5394743/5394743 - 11s 2us/step - loss: 0.0790 - acc: 0.9049 - val_loss: 0.0797 - val_acc: 0.9046
Epoch 30/30
5394743/5394743 - 11s 2us/step - loss: 0.0785 - acc: 0.9049 - val_loss: 0.0793 - val_acc: 0.9046
Updating ModelResults...
        Creating Model/Mode...
Done
```

Out[4]:

Model	Mode	#Top10	%Top10	Accuracy	MSE	Time
Random	Val & Test	23.0	0.423183	NaN	NaN	NaN
Static-Binary-EmbeddingRegressor-CB	Train	NaN	NaN	0.904774	0.082767	278.341788
	Val & Test	7.0	0.128795	0.905013	0.083044	43.151123
Static-Binary-EmbeddingRegressor-CF	Train	NaN	NaN	0.904929	0.079320	259.256753
	Val & Test	51.0	0.938362	0.904836	0.079984	41.145598
Static-Binary-EmbeddingRegressor-CFCB	Train	NaN	NaN	0.904883	0.078476	297.840885

```
In [5]: min_index = history_train.history['val_loss'].index(min(history_train.history['val_loss']))

        import matplotlib.pyplot as plt
        plt.plot(history_train.history['loss'], label = "Train")
        plt.plot(history_train.history['val_loss'], label = "Validation")
        plt.axvline(x=min_index, color = "red", label = "Best Model")
        plt.title("Train "+dic["Model"])
        plt.ylabel('MSE')
        plt.xlabel('# Epoch')
        plt.legend()
        plt.savefig("./graphs/"+ dic["Model"], dpi = 300)
        plt.show()
```



B.4. Test

```
In [6]: # Carga de datos para test
        df_real_test = pd.read_csv("../../Data/targetTest.dat")
        test_user = pd.read_csv("../../Data/userTest.corregido.dat", names = ["user"])
        test_item = pd.read_csv("../../Data/maestroTarget.dat")
        df_atributes_unmaped = pd.read_csv("../../Data/maestro.dat")

        # Generación de combinatoria de user-item propuestos
        test = pd.DataFrame.from_records(list(i for i in product(test_user["user"],
                                                                 test_item["item"])),
                                         columns=['user' , 'item'])
        test = test.merge(df_atributes_unmaped,how='left', left_on = "item", right_on= "item")


        # Mapeado de usuarios e items para el DataFrame de combinatoria
```

```
        df_user_map = pd.read_pickle("../../ProcessedData/UserMap")
        df_item_map = pd.read_pickle("../../ProcessedData/ItemMap")
        df_item_atr1_map = pd.read_pickle("../../ProcessedData/ItemAtribute1Map")
        df_item_atr2_map = pd.read_pickle("../../ProcessedData/ItemAtribute2Map")

        test = test.merge(df_user_map,how='left', left_on = "user", right_on= "Old_User_Id")
        test = test.merge(df_item_map,how='left', left_on = "item", right_on= "Old_Item_Id")
        test = test.merge(df_item_atr1_map,how='left', left_on = "atributo1",
                          right_on= "Old_Atribute1")
        test = test.merge(df_item_atr2_map,how='left', left_on = "atributo2",
                          right_on= "Old_Atribute2")
        test = test.drop(["atributo1", "atributo2",
                          "Old_User_Id", "Old_Item_Id",
                          "Old_Atribute1", "Old_Atribute2"], axis = 1)

        test = test.rename({"user":"unmapped_user", "item":"unmapped_item",
                            "New_User_Id":"user", "New_Item_Id":"item",
                            "New_Atribute1":"atributo1", "New_Atribute2":"atributo2"}, axis=1)
        test = test.fillna(0)

In [7]: model = buildModel(train)
        path_model = "./Trained/Static-Binary-EmbeddingRegressor-CFCB"

        model.load_weights(path_model)

        start_time = time.time()
        test["y_pred"] = model.predict(x=[test["user"], test["item"],
                                          test["atributo1"], test["atributo2"]], verbose=0)
        delta_time_test = time.time() - start_time

        test = test[["unmapped_user", "unmapped_item", "y_pred"]]
        test = test.rename({"unmapped_user":"user",
                            "unmapped_item":"item",}, axis=1)

        df_top_10 = test.groupby('user')['item', 'y_pred']
        temp = df_top_10.apply(lambda x: x.sort_values("y_pred",
                                                       ascending=False).head(10)).reset_index()
        df_top_10 = temp.drop(["level_1", "y_pred"], axis = 1)

        num_top_10 = score_accenture(df_real_test, df_top_10)

        dic = {
            "Model": "Static-Binary-EmbeddingRegressor-CFCB",
            "Mode": "Val & Test",
            "Time": delta_time_test,
            "MSE": min(history_train.history["val_loss"]),
            "Accuracy": max(history_train.history["val_acc"]),
            "#Top10" : num_top_10,
            "%Top10" : 100*num_top_10/df_real_test.shape[0],
        }

        df_results = save_results(dic)
        df_results

Updating ModelResults...
        Creating Model/Mode...
```

```
Done
```

Out[7]:

Model	Mode	#Top10	%Top10	Accuracy	MSE	Time
Random	Val & Test	23.0	0.423183	NaN	NaN	NaN
Static-Binary-EmbeddingRegressor-CB	Train	NaN	NaN	0.904774	0.082767	278.341788
	Val & Test	7.0	0.128795	0.905013	0.083044	43.151123
Static-Binary-EmbeddingRegressor-CF	Train	NaN	NaN	0.904929	0.079320	259.256753
	Val & Test	51.0	0.938362	0.904836	0.079984	41.145598
Static-Binary-EmbeddingRegressor-CFCB	Train	NaN	NaN	0.904883	0.078476	297.840885
	Val & Test	44.0	0.809568	0.904594	0.079316	45.660820

C

Ejemplo de modelo predictivo de relevancia gradual

C.1. Imports

```
In [1]: from keras.layers import Input, Embedding, Dot, Reshape, Dense
        from keras.layers import Flatten, concatenate, Dropout, BatchNormalization
        from keras.models import Model
        from keras.callbacks import ModelCheckpoint
        from keras.optimizers import Adam
        from keras.utils import plot_model
        import pandas as pd
        import numpy as np
        import random
        import time
        from itertools import product

        import os,sys,inspect
        currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.currentframe())))
        parentdir = os.path.dirname(currentdir)
        sys.path.insert(0,parentdir)

        from OwnCodeAndMetrics import *

        %matplotlib inline
        np.random.seed(123)
```

```
Using TensorFlow backend.
```

C.2. Function BuildModel

```
In [2]: def buildModel(df_user_item):

            num_users = len(df_user_item["user"].unique())+1
            num_items = len(df_user_item["item"].unique())+1
            num_atr1 = len(df_user_item["atributo1"].unique())+1
            num_atr2 = len(df_user_item["atributo2"].unique())+1

            embedding_cf_size = 10
            atrib_embedding_size = 4
```

```
        # All inputs are 1-dimensional
        user = Input(name = 'user', shape = [1])
        item = Input(name = 'item', shape = [1])
        atr1 = Input(name = 'atr1', shape = [1])
        atr2 = Input(name = 'atr2', shape = [1])


        # Embedding the user (shape will be (None, 1, embedding_size))
        user_embedding = Embedding(name = 'user_embedding_to_dot',
                                   input_dim = num_users,
                                   output_dim = embedding_cf_size)(user)

        # Embedding the item (shape will be (None, 1, embedding_size))
        item_embedding = Embedding(name = 'item_embedding_to_dot',
                                   input_dim = num_items,
                                   output_dim = embedding_cf_size)(item)

        # Embedding the item atribute 1 (shape will be (None, 1, embedding_size))
        atr1_embedding = Embedding(name = 'atr1_embedding',
                                   input_dim = num_atr1,
                                   output_dim = atrib_embedding_size)(atr1)

        # Embedding the item atribute 2 (shape will be (None, 1, embedding_size))
        atr2_embedding = Embedding(name = 'item_atr2_embedding',
                                   input_dim = num_atr2,
                                   output_dim = atrib_embedding_size)(atr2)

        user_flat = Flatten()(user_embedding)
        item_flat = Flatten()(item_embedding)
        atr1_flat = Flatten()(atr1_embedding)
        atr2_flat = Flatten()(atr2_embedding)

        # Concatenate the Embedding layers
        conc = concatenate([(user_flat),
                            (item_flat),
                            (atr1_flat),
                            (atr2_flat)])


        d1 =    Dense(64, activation='relu', name = "Dense1")(conc)

        out =   Dense(1)(d1)

        model = Model(inputs = [user, item, atr1, atr2], outputs = out)
        # Compile using specified optimizer and loss
        model.compile(optimizer = Adam(lr=0.001), loss = 'mse', metrics = ["acc"])

        return model
```

C.2.1. Loading Data

```
In [3]: train = pd.read_pickle("../../ProcessedData/(Total)Static-UserItem-MatrixWeighted")
        items = pd.read_pickle("../../ProcessedData/ItemContent")
        train = train.merge(items,how='left', left_on = "item", right_on= "item")
```

```
train = train.sample(frac=1).reset_index(drop=True)
```

C.3. Train

```
In [4]: #gen = generate_batch(train, batch_size)

        batch_size = 1000000
        path_model = "./Trained/Static-Weighted-EmbeddingRegressor-CFCB"
        mcp = ModelCheckpoint(path_model, monitor="val_loss",
                              save_best_only=True, save_weights_only=False)

        # Train
        model = buildModel(train)
        print(model.summary())

        start_time = time.time()
        history_train = model.fit([train["user"], train["item"],
                                   train["atributo1"], train["atributo2"]]
                                   , train["rel"]
                                   , batch_size=batch_size, epochs=40
                                   , validation_split=0.1
                                   , callbacks=[mcp]
                                   , verbose = 1)
        delta_time_train = time.time() - start_time

        dic = {
            "Model": "Static-Weighted-EmbeddingRegressor-CFCB",
            "Mode": "Train",
            "Time": delta_time_train,
            "MSE": min(history_train.history["loss"]),
            "Accuracy": max(history_train.history["acc"])
        }
        df_results = save_results(dic)
        df_results
```

```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
user (InputLayer)               (None, 1)            0
__________________________________________________________________________________________________
item (InputLayer)               (None, 1)            0
__________________________________________________________________________________________________
atr1 (InputLayer)               (None, 1)            0
__________________________________________________________________________________________________
atr2 (InputLayer)               (None, 1)            0
__________________________________________________________________________________________________
user_embedding_to_dot (Embeddin (None, 1, 10)        2355770     user[0][0]
__________________________________________________________________________________________________
item_embedding_to_dot (Embeddin (None, 1, 10)        77810       item[0][0]
__________________________________________________________________________________________________
atr1_embedding (Embedding)      (None, 1, 4)         16          atr1[0][0]
__________________________________________________________________________________________________
item_atr2_embedding (Embedding) (None, 1, 4)         84          atr2[0][0]
```

```
____________________________________________________________________________________________________
flatten_1 (Flatten)              (None, 10)            0           user_embedding_to_dot[0][0]
____________________________________________________________________________________________________
flatten_2 (Flatten)              (None, 10)            0           item_embedding_to_dot[0][0]
____________________________________________________________________________________________________
flatten_3 (Flatten)              (None, 4)             0           atr1_embedding[0][0]
____________________________________________________________________________________________________
flatten_4 (Flatten)              (None, 4)             0           item_atr2_embedding[0][0]
____________________________________________________________________________________________________
concatenate_1 (Concatenate)      (None, 28)            0           flatten_1[0][0]
                                                                   flatten_2[0][0]
                                                                   flatten_3[0][0]
                                                                   flatten_4[0][0]
____________________________________________________________________________________________________
Dense1 (Dense)                   (None, 64)            1856        concatenate_1[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 1)             65          Dense1[0][0]
====================================================================================================
Total params: 2,435,601
Trainable params: 2,435,601
Non-trainable params: 0
____________________________________________________________________________________________________
None
Train on 5394743 samples, validate on 599416 samples
Epoch 1/40
5394743/5394743 - 10s 2us/step - loss: 0.0258 - acc: 0.6217 - val_loss: 0.0250 - val_acc: 0.6212
Epoch 2/40
5394743/5394743 - 10s 2us/step - loss: 0.0251 - acc: 0.6217 - val_loss: 0.0246 - val_acc: 0.6212
Epoch 3/40
5394743/5394743 - 10s 2us/step - loss: 0.0246 - acc: 0.6217 - val_loss: 0.0243 - val_acc: 0.6212
Epoch 4/40
5394743/5394743 - 10s 2us/step - loss: 0.0243 - acc: 0.6217 - val_loss: 0.0239 - val_acc: 0.6212
Epoch 5/40
5394743/5394743 - 10s 2us/step - loss: 0.0239 - acc: 0.6217 - val_loss: 0.0237 - val_acc: 0.6212
Epoch 6/40
5394743/5394743 - 10s 2us/step - loss: 0.0237 - acc: 0.6217 - val_loss: 0.0234 - val_acc: 0.6212
Epoch 7/40
5394743/5394743 - 9s 2us/step - loss: 0.0234 - acc: 0.6217 - val_loss: 0.0232 - val_acc: 0.6212
Epoch 8/40
5394743/5394743 - 10s 2us/step - loss: 0.0231 - acc: 0.6217 - val_loss: 0.0229 - val_acc: 0.6212
Epoch 9/40
5394743/5394743 - 10s 2us/step - loss: 0.0227 - acc: 0.6217 - val_loss: 0.0226 - val_acc: 0.6212
Epoch 10/40
5394743/5394743 - 10s 2us/step - loss: 0.0224 - acc: 0.6217 - val_loss: 0.0223 - val_acc: 0.6212
Epoch 11/40
5394743/5394743 - 9s 2us/step - loss: 0.0219 - acc: 0.6217 - val_loss: 0.0219 - val_acc: 0.6212
Epoch 12/40
5394743/5394743 - 10s 2us/step - loss: 0.0214 - acc: 0.6217 - val_loss: 0.0215 - val_acc: 0.6212
Epoch 13/40
5394743/5394743 - 9s 2us/step - loss: 0.0209 - acc: 0.6218 - val_loss: 0.0212 - val_acc: 0.6213
Epoch 14/40
5394743/5394743 - 10s 2us/step - loss: 0.0203 - acc: 0.6220 - val_loss: 0.0208 - val_acc: 0.6215
Epoch 15/40
5394743/5394743 - 9s 2us/step - loss: 0.0196 - acc: 0.6221 - val_loss: 0.0204 - val_acc: 0.6216
Epoch 16/40
5394743/5394743 - 9s 2us/step - loss: 0.0189 - acc: 0.6223 - val_loss: 0.0200 - val_acc: 0.6218
Epoch 17/40
```

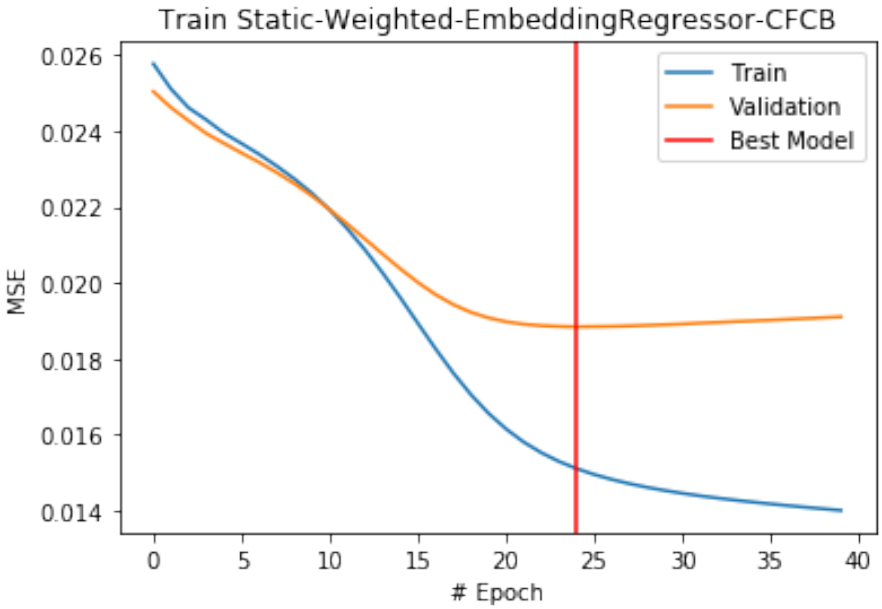
```
5394743/5394743 - 9s 2us/step - loss: 0.0183 - acc: 0.6226 - val_loss: 0.0197 - val_acc: 0.6221
Epoch 18/40
5394743/5394743 - 10s 2us/step - loss: 0.0176 - acc: 0.6230 - val_loss: 0.0194 - val_acc: 0.6224
Epoch 19/40
5394743/5394743 - 10s 2us/step - loss: 0.0171 - acc: 0.6235 - val_loss: 0.0192 - val_acc: 0.6226
Epoch 20/40
5394743/5394743 - 9s 2us/step - loss: 0.0166 - acc: 0.6239 - val_loss: 0.0191 - val_acc: 0.6227
Epoch 21/40
5394743/5394743 - 10s 2us/step - loss: 0.0162 - acc: 0.6243 - val_loss: 0.0190 - val_acc: 0.6229
Epoch 22/40
5394743/5394743 - 9s 2us/step - loss: 0.0158 - acc: 0.6246 - val_loss: 0.0189 - val_acc: 0.6230
Epoch 23/40
5394743/5394743 - 10s 2us/step - loss: 0.0155 - acc: 0.6249 - val_loss: 0.0189 - val_acc: 0.6230
Epoch 24/40
5394743/5394743 - 10s 2us/step - loss: 0.0153 - acc: 0.6251 - val_loss: 0.0189 - val_acc: 0.6230
Epoch 25/40
5394743/5394743 - 10s 2us/step - loss: 0.0151 - acc: 0.6252 - val_loss: 0.0188 - val_acc: 0.6230
Epoch 26/40
5394743/5394743 - 11s 2us/step - loss: 0.0149 - acc: 0.6254 - val_loss: 0.0188 - val_acc: 0.6230
Epoch 27/40
5394743/5394743 - 9s 2us/step - loss: 0.0148 - acc: 0.6255 - val_loss: 0.0189 - val_acc: 0.6230
Epoch 28/40
5394743/5394743 - 9s 2us/step - loss: 0.0147 - acc: 0.6255 - val_loss: 0.0189 - val_acc: 0.6230
Epoch 29/40
5394743/5394743 - 9s 2us/step - loss: 0.0146 - acc: 0.6256 - val_loss: 0.0189 - val_acc: 0.6231
Epoch 30/40
5394743/5394743 - 9s 2us/step - loss: 0.0145 - acc: 0.6257 - val_loss: 0.0189 - val_acc: 0.6231
Epoch 31/40
5394743/5394743 - 9s 2us/step - loss: 0.0145 - acc: 0.6257 - val_loss: 0.0189 - val_acc: 0.6230
Epoch 32/40
5394743/5394743 - 9s 2us/step - loss: 0.0144 - acc: 0.6258 - val_loss: 0.0189 - val_acc: 0.6230
Epoch 33/40
5394743/5394743 - 9s 2us/step - loss: 0.0143 - acc: 0.6258 - val_loss: 0.0190 - val_acc: 0.6230
Epoch 34/40
5394743/5394743 - 9s 2us/step - loss: 0.0143 - acc: 0.6258 - val_loss: 0.0190 - val_acc: 0.6230
Epoch 35/40
5394743/5394743 - 10s 2us/step - loss: 0.0142 - acc: 0.6258 - val_loss: 0.0190 - val_acc: 0.6229
Epoch 36/40
5394743/5394743 - 10s 2us/step - loss: 0.0142 - acc: 0.6259 - val_loss: 0.0190 - val_acc: 0.6229
Epoch 37/40
5394743/5394743 - 11s 2us/step - loss: 0.0141 - acc: 0.6259 - val_loss: 0.0190 - val_acc: 0.6229
Epoch 38/40
5394743/5394743 - 9s 2us/step - loss: 0.0141 - acc: 0.6259 - val_loss: 0.0191 - val_acc: 0.6229
Epoch 39/40
5394743/5394743 - 10s 2us/step - loss: 0.0140 - acc: 0.6259 - val_loss: 0.0191 - val_acc: 0.6228
Epoch 40/40
5394743/5394743 - 10s 2us/step - loss: 0.0140 - acc: 0.6259 - val_loss: 0.0191 - val_acc: 0.6228
Updating ModelResults...
        Creating Model/Mode...
Done
```

Out[4]:

			#Top10	%Top10
	Model	Mode		
	Random	Val & Test	23.0	0.423183
	Static-Binary-EmbeddingRegressor-CB	Train	NaN	NaN

```
                                          Val & Test     7.0  0.128795
Static-Binary-EmbeddingRegressor-CF       Train          NaN       NaN
                                          Val & Test    51.0  0.938362
Static-Binary-EmbeddingRegressor-CFCB     Train          NaN       NaN
                                          Val & Test    44.0  0.809568
Static-Weighted-EmbeddingRegressor-CB     Train          NaN       NaN
                                          Val & Test    11.0  0.202392
Static-Weighted-EmbeddingRegressor-CF     Train          NaN       NaN
                                          Val & Test    20.0  0.367985
Static-Weighted-EmbeddingRegressor-CFCB   Train          NaN       NaN

                                                      Accuracy       MSE  \
Model                                     Mode
Random                                    Val & Test       NaN       NaN
Static-Binary-EmbeddingRegressor-CB       Train       0.904774  0.082767
                                          Val & Test  0.905013  0.083044
Static-Binary-EmbeddingRegressor-CF       Train       0.904929  0.079320
                                          Val & Test  0.904836  0.079984
Static-Binary-EmbeddingRegressor-CFCB     Train       0.904883  0.078476
                                          Val & Test  0.904594  0.079316
Static-Weighted-EmbeddingRegressor-CB     Train       0.624957  0.016196
                                          Val & Test  0.622481  0.020623
Static-Weighted-EmbeddingRegressor-CF     Train       0.625827  0.014210
                                          Val & Test  0.623138  0.018811
Static-Weighted-EmbeddingRegressor-CFCB   Train       0.625915  0.014003

                                                          Time
Model                                     Mode
Random                                    Val & Test       NaN
Static-Binary-EmbeddingRegressor-CB       Train       278.341788
                                          Val & Test   43.151123
Static-Binary-EmbeddingRegressor-CF       Train       259.256753
                                          Val & Test   41.145598
Static-Binary-EmbeddingRegressor-CFCB     Train       297.840885
                                          Val & Test   45.660820
Static-Weighted-EmbeddingRegressor-CB     Train       358.350441
                                          Val & Test   45.928401
Static-Weighted-EmbeddingRegressor-CF     Train       344.906921
                                          Val & Test   39.594234
Static-Weighted-EmbeddingRegressor-CFCB   Train       387.318596
```

```
In [5]: min_index = history_train.history['val_loss'].index(min(history_train.history['val_loss']))

        import matplotlib.pyplot as plt
        plt.plot(history_train.history['loss'], label = "Train")
        plt.plot(history_train.history['val_loss'], label = "Validation")
        plt.axvline(x=min_index, color = "red", label = "Best Model")
        plt.title("Train "+dic["Model"])
        plt.ylabel('MSE')
        plt.xlabel('# Epoch')
        plt.legend()
        plt.savefig("./graphs/"+ dic["Model"], dpi = 300)
        plt.show()
```



C.4. Test

```
In [6]: # Carga de datos para test
        df_real_test = pd.read_csv("../../Data/targetTest.dat")
        test_user = pd.read_csv("../../Data/userTest.corregido.dat", names = ["user"])
        test_item = pd.read_csv("../../Data/maestroTarget.dat")
        df_atributes_unmaped = pd.read_csv("../../Data/maestro.dat")

        # Generación de combinatoria de user-item propuestos
        test = pd.DataFrame.from_records(list(i for i in product(test_user["user"],
                                                                 test_item["item"])),
                                         columns=['user' , 'item'])
        test = test.merge(df_atributes_unmaped,how='left', left_on = "item", right_on= "item")


        # Mapeado de usuarios e items para el DataFrame de combinatoria
        df_user_map = pd.read_pickle("../../ProcessedData/UserMap")
        df_item_map = pd.read_pickle("../../ProcessedData/ItemMap")
        df_item_atr1_map = pd.read_pickle("../../ProcessedData/ItemAtribute1Map")
        df_item_atr2_map = pd.read_pickle("../../ProcessedData/ItemAtribute2Map")

        test = test.merge(df_user_map,how='left', left_on = "user", right_on= "Old_User_Id")
        test = test.merge(df_item_map,how='left', left_on = "item", right_on= "Old_Item_Id")
        test = test.merge(df_item_atr1_map,how='left', left_on = "atributo1",
                          right_on= "Old_Atribute1")
        test = test.merge(df_item_atr2_map,how='left', left_on = "atributo2",
                          right_on= "Old_Atribute2")
        test = test.drop(["atributo1", "atributo2",
                          "Old_User_Id", "Old_Item_Id",
```

```
                          "Old_Atribute1", "Old_Atribute2"], axis = 1)

        test = test.rename({"user":"unmapped_user", "item":"unmapped_item",
                            "New_User_Id":"user", "New_Item_Id":"item",
                            "New_Atribute1":"atributo1", "New_Atribute2":"atributo2"}, axis=1)
        test = test.fillna(0)

In [7]: model = buildModel(train)
        path_model = "./Trained/Static-Weighted-EmbeddingRegressor-CFCB"

        model.load_weights(path_model)

        start_time = time.time()
        test["y_pred"] = model.predict(x=[test["user"], test["item"],
                                          test["atributo1"], test["atributo2"]], verbose=0)
        delta_time_test = time.time() - start_time

        test = test[["unmapped_user", "unmapped_item", "y_pred"]]
        test = test.rename({"unmapped_user":"user",
                        "unmapped_item":"item",}, axis=1)

        df_top_10 = test.groupby('user')['item', 'y_pred']
        temp = df_top_10.apply(lambda x: x.sort_values("y_pred",
                                                       ascending=False).head(10)).reset_index()
        df_top_10 = temp.drop(["level_1", "y_pred"], axis = 1)

        num_top_10 = score_accenture(df_real_test, df_top_10)

        dic = {
            "Model": "Static-Weighted-EmbeddingRegressor-CFCB",
            "Mode": "Val & Test",
            "Time": delta_time_test,
            "MSE": min(history_train.history["val_loss"]),
            "Accuracy": max(history_train.history["val_acc"]),
            "#Top10" : num_top_10,
            "%Top10" : 100*num_top_10/df_real_test.shape[0],
        }

        df_results = save_results(dic)
        df_results

Updating ModelResults...
        Creating Model/Mode...
Done
```

Out[7]:

		#Top10	%Top10
Model	Mode		
Random	Val & Test	23.0	0.423183
Static-Binary-EmbeddingRegressor-CB	Train	NaN	NaN
	Val & Test	7.0	0.128795
Static-Binary-EmbeddingRegressor-CF	Train	NaN	NaN
	Val & Test	51.0	0.938362
Static-Binary-EmbeddingRegressor-CFCB	Train	NaN	NaN
	Val & Test	44.0	0.809568
Static-Weighted-EmbeddingRegressor-CB	Train	NaN	NaN

```
                                        Val & Test    11.0  0.202392
Static-Weighted-EmbeddingRegressor-CF   Train          NaN       NaN
                                        Val & Test    20.0  0.367985
Static-Weighted-EmbeddingRegressor-CFCB Train          NaN       NaN
                                        Val & Test    24.0  0.441582

                                                  Accuracy       MSE  \
Model                                   Mode
Random                                  Val & Test       NaN       NaN
Static-Binary-EmbeddingRegressor-CB     Train       0.904774  0.082767
                                        Val & Test  0.905013  0.083044
Static-Binary-EmbeddingRegressor-CF     Train       0.904929  0.079320
                                        Val & Test  0.904836  0.079984
Static-Binary-EmbeddingRegressor-CFCB   Train       0.904883  0.078476
                                        Val & Test  0.904594  0.079316
Static-Weighted-EmbeddingRegressor-CB   Train       0.624957  0.016196
                                        Val & Test  0.622481  0.020623
Static-Weighted-EmbeddingRegressor-CF   Train       0.625827  0.014210
                                        Val & Test  0.623138  0.018811
Static-Weighted-EmbeddingRegressor-CFCB Train       0.625915  0.014003
                                        Val & Test  0.623105  0.018841

                                                        Time
Model                                   Mode
Random                                  Val & Test       NaN
Static-Binary-EmbeddingRegressor-CB     Train       278.341788
                                        Val & Test   43.151123
Static-Binary-EmbeddingRegressor-CF     Train       259.256753
                                        Val & Test   41.145598
Static-Binary-EmbeddingRegressor-CFCB   Train       297.840885
                                        Val & Test   45.660820
Static-Weighted-EmbeddingRegressor-CB   Train       358.350441
                                        Val & Test   45.928401
Static-Weighted-EmbeddingRegressor-CF   Train       344.906921
                                        Val & Test   39.594234
Static-Weighted-EmbeddingRegressor-CFCB Train       387.318596
                                        Val & Test   48.826483
```

D

Ejemplo de ejecución de agente basado dn Deep RL

D.1. Imports

```
In [12]: %reload_ext autoreload
         %autoreload 2
         %matplotlib inline

         import nbimporter
         from Visualizator import *
         import argparse
         import sys
         from PIL import Image
         import numpy as np
         import gym, reco_gym
         from reco_gym import env_1_args
         from keras.models import *
         from keras.layers import *
         from keras.optimizers import Adam
         from keras.regularizers import l2
         import keras.backend as K
         from rl.agents.dqn import *
         from rl.agents import *
         from rl.policy import *
         from rl.memory import *
         from rl.core import Processor
         from rl.util import load_demo_data_from_file
         from record_demonstrations import demonstrate #Github externo
         import matplotlib.pyplot as plt
         import pandas as pd
         from copy import deepcopy
         import tensorflow as tf
         import logging
         from record_demonstrations import *
         tf.get_logger().setLevel(logging.ERROR)
```

D.2. Classes and Functions needed

```
In [13]: class RecoProcessor(Processor):
             def __init__(self, look_back):
```

```
            self.look_back = look_back

        def process_observation(self, observation):
            if observation is None:
                X=np.zeros(self.look_back)
            else:
                if len(observation)>self.look_back:
                    observation = observation[-self.look_back]
                observation = np.array(observation)
                if len(observation.shape) == 2:
                    X = observation[:,1]
                else:
                    X = np.array([observation[1]])
            if len(X)<self.look_back:
                X = np.append(X,np.zeros(self.look_back-len(X)))
            return X

        def process_state_batch(self, batch):
            return np.array(batch).astype('float32')

        def process_reward(self, reward):
            return reward

        def process_demo_data(self, demo_data):
            for step in demo_data:
                step[0] = self.process_observation(step[0])
                step[2] = self.process_reward(step[2])
            return demo_data

    """
    Huber loss for Q Learning
    References: https://en.wikipedia.org/wiki/Huber_loss
                https://www.tensorflow.org/api_docs/python/tf/losses/huber_loss
    """
    def huber_loss(y_true, y_pred, clip_delta=1.0):
        error = y_true - y_pred
        cond  = K.abs(error) <= clip_delta

        squared_loss = 0.5 * K.square(error)
        quadratic_loss = 0.5 * K.square(clip_delta) + clip_delta * (K.abs(error) - clip_delta)

        return K.mean(tf.where(cond, squared_loss, quadratic_loss))
```

D.3. Defining the environment

```
In [14]: env_1_args['random_seed'] = 42
         env_1_args['num_products'] = 100

         ENV_NAME = "reco-gym-v1"
         env = gym.make(ENV_NAME)
         env.init_gym(env_1_args)

         np.random.seed(123)
         env.reset()
```

D.4. Creating the Agent

Definition

```
In [15]: lr = 1e-3
         directorio = "./outfiles/"
         modelo = "DoubleDDQN"
         fichero_modelo = "./trained_models/"+modelo+"_model_"+ENV_NAME+".h5f"
         fichero_train =  directorio+""+modelo+"_train_"+ENV_NAME+".txt"
         fichero_test =   directorio+""+modelo+"_test_"+ENV_NAME+".txt"
         wl = 1
         emb_size = 10
         look_back = 1
         nb_steps = 1000000

         # model
         inp = Input(shape=(1,)+(look_back, ))
         emb = Embedding(input_dim = env.action_space.n+1,output_dim = emb_size)(inp)
         fla = Flatten()(emb)
         de1 = Dense(64, activation='relu')(fla)
         de2 = Dense(64, activation='relu')(de1)
         out = Dense(env.action_space.n, activation='softmax')(de2)

         model = Model(inputs=inp, outputs=out)
         model.compile(loss=huber_loss, optimizer=Adam(lr=lr))


         # memory
         memory = PrioritizedMemory(limit=500000, alpha=.6, start_beta=.4,
                                    end_beta=.4, window_length=wl)

         # policy
         policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps',
                                       value_max=1., value_min=.1, value_test=.05,
                                       nb_steps=int(nb_steps/5 * 4))

         # processor
         processor = RecoProcessor(look_back)

         # agent
         agent = DQNAgent(model=model, nb_actions=env.action_space.n,
                          memory=memory, processor=processor,
                          nb_steps_warmup=10,enable_double_dqn=True, enable_dueling_network=True,
                          gamma=.999, target_model_update=lr/10.0, policy=policy)
         agent.compile(optimizer=Adam(lr=lr))
```

Training

```
In [5]: train = agent.fit(env, nb_steps=nb_steps, visualize=False, verbose=1,
                          nb_max_start_steps=1, log_interval=100000,
                          start_step_policy=(lambda obs: None), # First action must be None.
                                                                # rl.core changed from U[0,1] to 1
                                                                # if nb_max_start_steps != 0
                          nb_max_episode_steps = None #Run until done == True)
```

```
                    )
        agent.save_weights(fichero_modelo, overwrite = True)
        np.savetxt(fichero_train, np.array(train.history["episode_reward"]), delimiter=",")
```

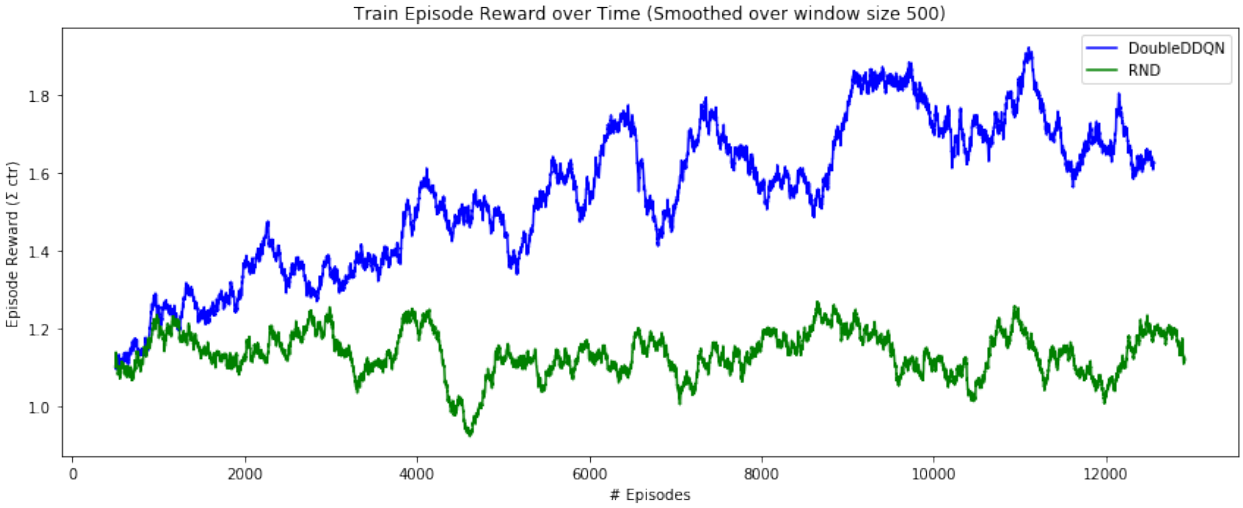
```
Training for 1000000 steps ...
Interval 1 (0 steps performed)
100000/100000 - 1351s 14ms/step - reward: 0.0153
1286 episodes - episode_reward: 1.187 [0.000, 10.000] - loss: 0.000 - mean_q: 0.229 - mean_eps: 0.944

Interval 2 (100000 steps performed)
100000/100000 - 1401s 14ms/step - reward: 0.0167
1253 episodes - episode_reward: 1.330 [0.000, 11.000] - loss: 0.000 - mean_q: 0.485 - mean_eps: 0.831

Interval 3 (200000 steps performed)
100000/100000 - 1270s 13ms/step - reward: 0.0179
1289 episodes - episode_reward: 1.392 [0.000, 16.000] - loss: 0.001 - mean_q: 0.687 - mean_eps: 0.719

Interval 4 (300000 steps performed)
100000/100000 - 702s 7ms/step - reward: 0.0185
1268 episodes - episode_reward: 1.462 [0.000, 13.000] - loss: 0.001 - mean_q: 0.860 - mean_eps: 0.606

Interval 5 (400000 steps performed)
100000/100000 - 709s 7ms/step - reward: 0.0199
1216 episodes - episode_reward: 1.637 [0.000, 13.000] - loss: 0.001 - mean_q: 1.001 - mean_eps: 0.494

Interval 6 (500000 steps performed)
100000/100000 - 724s 7ms/step - reward: 0.0204
1258 episodes - episode_reward: 1.622 [0.000, 14.000] - loss: 0.010 - mean_q: 1.122 - mean_eps: 0.381

Interval 7 (600000 steps performed)
100000/100000 - 727s 7ms/step - reward: 0.0199
1254 episodes - episode_reward: 1.585 [0.000, 15.000] - loss: 0.011 - mean_q: 1.219 - mean_eps: 0.269

Interval 8 (700000 steps performed)
100000/100000 - 743s 7ms/step - reward: 0.0227
1280 episodes - episode_reward: 1.779 [0.000, 15.000] - loss: 0.012 - mean_q: 1.297 - mean_eps: 0.156

Interval 9 (800000 steps performed)
100000/100000 - 580s 6ms/step - reward: 0.0214
1225 episodes - episode_reward: 1.752 [0.000, 18.000] - loss: 0.013 - mean_q: 1.354 - mean_eps: 0.100

Interval 10 (900000 steps performed)
100000/100000 - 514s 5ms/step - reward: 0.0204
done, took 8723.072 seconds
```

Testing

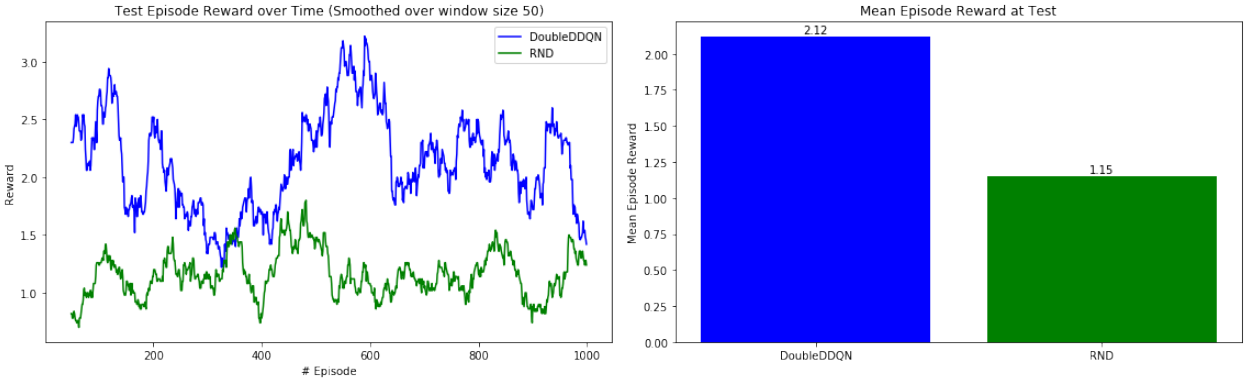
```
In [6]: agent.load_weights(fichero_modelo)
        test = agent.test(env, nb_episodes=1000, visualize=False, verbose=0)
        np.savetxt(fichero_test, np.array(test.history["episode_reward"]), delimiter=",")
```

Plots

```
In [10]: train_comparison(models=[modelo, "RND"], outfile = "Train"+modelo+".png")
```




```
In [11]: test_comparison(models=[modelo, "RND"], outfile = "Test"+modelo+".png")
```



Creating demonstrations of expert model

```
In [16]: agent.load_weights(fichero_modelo)
         demonstrate(agent, env, 75000, "./demos/"+str(ENV_NAME)+'demos_'+modelo+'.npy')
```

Ejemplo de ejecución de agente basado en Deep RL from DEMONSTRATIONS

E.1. Imports

In [1]:
```
%reload_ext autoreload
%autoreload 2
%matplotlib inline

import nbimporter
from Visualizator import *
import argparse
import sys
from PIL import Image
import numpy as np
import gym, reco_gym
from reco_gym import env_1_args
from keras.models import *
from keras.layers import *
from keras.optimizers import Adam
from keras.regularizers import l2
import keras.backend as K
from rl.agents.dqn import *
from rl.agents import *
from rl.policy import *
from rl.memory import *
from rl.core import Processor
from rl.util import load_demo_data_from_file
import matplotlib.pyplot as plt
import pandas as pd
from copy import deepcopy
import tensorflow as tf
import logging
tf.get_logger().setLevel(logging.ERROR)
```

```
Importing Jupyter notebook from Visualizator.ipynb
```

```
Using TensorFlow backend.
```

E.2. Classes and Functions needed

```
In [2]: class RecoProcessor(Processor):
            def __init__(self, look_back):
                self.look_back = look_back

            def process_observation(self, observation):
                if observation is None:
                    X=np.zeros(self.look_back)
                else:
                    if len(observation)>self.look_back:
                        observation = observation[-self.look_back]
                    observation = np.array(observation)
                    if len(observation.shape) == 2:
                        X = observation[:,1]
                    else:
                        X = np.array([observation[1]])
                if len(X)<self.look_back:
                    X = np.append(X,np.zeros(self.look_back-len(X)))
                return X

            def process_state_batch(self, batch):
                return np.array(batch)

            def process_reward(self, reward):
                return reward

            def process_demo_data(self, demo_data):
                for step in demo_data:
                    step[0] = self.process_observation(step[0])
                    step[2] = self.process_reward(step[2])
                return demo_data

        """
        Huber loss for Q Learning
        References: https://en.wikipedia.org/wiki/Huber_loss
                    https://www.tensorflow.org/api_docs/python/tf/losses/huber_loss
        """
        def huber_loss(y_true, y_pred, clip_delta=1.0):
            error = y_true - y_pred
            cond  = K.abs(error) <= clip_delta

            squared_loss = 0.5 * K.square(error)
            quadratic_loss = 0.5 * K.square(clip_delta) + clip_delta * (K.abs(error) - clip_delta)

            return K.mean(tf.where(cond, squared_loss, quadratic_loss))
```

E.3. Defining the environment

```
In [3]: env_1_args['random_seed'] = 42
        env_1_args['num_products'] = 100

        ENV_NAME = "reco-gym-v1"
```

```
        env = gym.make(ENV_NAME)
        env.init_gym(env_1_args)

        np.random.seed(123)
        env.reset()
```

```
In [4]: prev_model = "DoubleDDQN"
        demo_file = "./demos/"+str(ENV_NAME)+'demos_'+prev_model+'.npy'
        demonstrations = load_demo_data_from_file(demo_file)
```

E.4. Creating the Agent

Definition

```
In [5]: lr = 1e-3
        directorio = "./outfiles/"
        modelo = "DoubleDDQNfD"
        fichero_modelo = "./trained_models/"+modelo+"_model_"+ENV_NAME+".h5f"
        fichero_train =  directorio+""+modelo+"_train_"+ENV_NAME+".txt"
        fichero_test =   directorio+""+modelo+"_test_"+ENV_NAME+".txt"
        wl = 1
        emb_size = 10
        look_back = 1
        nb_steps = 1000000

        # model
        inp = Input(shape=(1,)+(look_back, ))
        emb = Embedding(input_dim = env.action_space.n+1,output_dim = emb_size)(inp)
        fla = Flatten()(emb)
        de1 = Dense(64, activation='relu', kernel_regularizer=l2(.0001))(fla)
        de2 = Dense(64, activation='relu', kernel_regularizer=l2(.0001))(de1)
        out = Dense(env.action_space.n, activation='softmax', kernel_regularizer=l2(.0001))(de2)

        model = Model(inputs=inp, outputs=out)
        model.compile(loss=huber_loss, optimizer=Adam(lr=lr))

        # memory
        memory = PartitionedMemory(limit=500000,
                                   pre_load_data = demonstrations,
                                   alpha=.6, start_beta=.4, end_beta=.4,
                                   window_length=wl)

        # policy
        policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps',
                                      value_max=.7, value_min=.1, value_test=.05,
                                      nb_steps=int(nb_steps/5 * 4))

        # processor
        processor = RecoProcessor(look_back)

        # agent
        agent = DQfDAgent(model=model, nb_actions=env.action_space.n,
                          memory=memory, processor=processor,
```

```
                        nb_steps_warmup=10,enable_double_dqn=True, enable_dueling_network=True,
                        gamma=.999, target_model_update=lr/10.0, policy=policy, train_interval=1,
                        delta_clip=1., pretraining_steps=30000, n_step=10,large_margin=.8, lam_2=1)

        agent.compile(optimizer=Adam(lr=lr))
```

Training

```
In [6]: train = agent.fit(env, nb_steps=nb_steps, visualize=False, verbose=1,
                          nb_max_start_steps=1, log_interval = 100000,
                          start_step_policy=(lambda obs: None), # First action must be None.
                                                                # rl.core changed from U[0,1] to 1
                                                                # if nb_max_start_steps != 0
                          nb_max_episode_steps = None #Run until done == True)
                        )
        agent.save_weights(fichero_modelo, overwrite = True)
        np.savetxt(fichero_train, np.array(train.history["episode_reward"]), delimiter=",")
```

```
Pretraining for 30000 steps...
29997/30000 [============================>.] - ETA: 0s

Training for 1000000 steps ...
Interval 1 (0 steps performed)
100000/100000 - 1024s 10ms/step - reward: 0.0188
1276 episodes - episode_reward: 1.474 [0.000, 11.000] - loss: 0.035 - mean_q: 0.481 - mean_eps: 0.663

Interval 2 (100000 steps performed)
100000/100000 - 1085s 11ms/step - reward: 0.0200
1307 episodes - episode_reward: 1.534 [0.000, 16.000] - loss: 0.035 - mean_q: 0.826 - mean_eps: 0.588

Interval 3 (200000 steps performed)
100000/100000 - ETA: 0s - reward: 0.0210- ETA: 0s - r - 1011s 10ms/step - reward: 0.0210
1242 episodes - episode_reward: 1.692 [0.000, 18.000] - loss: 0.037 - mean_q: 1.034 - mean_eps: 0.513

Interval 4 (300000 steps performed)
100000/100000 - 1004s 10ms/step - reward: 0.0214
1266 episodes - episode_reward: 1.686 [0.000, 12.000] - loss: 0.040 - mean_q: 1.188 - mean_eps: 0.438

Interval 5 (400000 steps performed)
100000/100000 - 1019s 10ms/step - reward: 0.0233
1259 episodes - episode_reward: 1.853 [0.000, 13.000] - loss: 0.042 - mean_q: 1.320 - mean_eps: 0.363

Interval 6 (500000 steps performed)
100000/100000 - 1017s 10ms/step - reward: 0.0232
1307 episodes - episode_reward: 1.781 [0.000, 20.000] - loss: 0.045 - mean_q: 1.445 - mean_eps: 0.288

Interval 7 (600000 steps performed)
100000/100000 - 1020s 10ms/step - reward: 0.0251
1230 episodes - episode_reward: 2.039 [0.000, 19.000] - loss: 0.048 - mean_q: 1.559 - mean_eps: 0.213

Interval 8 (700000 steps performed)
100000/100000 - 1141s 11ms/step - reward: 0.0270
1312 episodes - episode_reward: 2.054 [0.000, 17.000] - loss: 0.050 - mean_q: 1.661 - mean_eps: 0.138

Interval 9 (800000 steps performed)
```

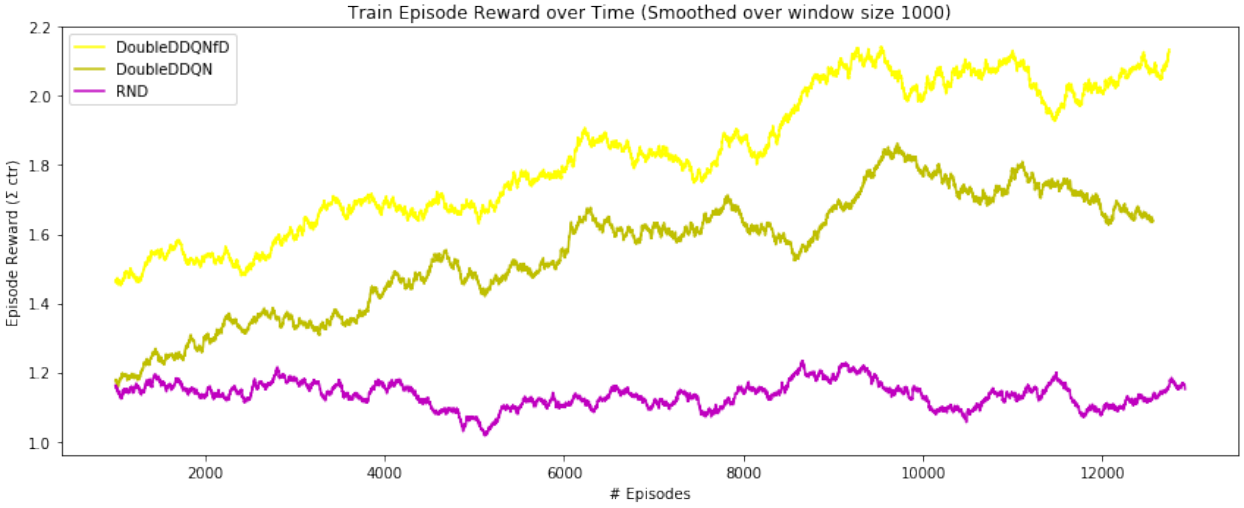
```
100000/100000 - 1334s 13ms/step - reward: 0.0254
1283 episodes - episode_reward: 1.978 [0.000, 14.000] - loss: 0.052 - mean_q: 1.759 - mean_eps: 0.100

Interval 10 (900000 steps performed)
100000/100000 - 1160s 12ms/step - reward: 0.0270
done, took 10816.419 seconds
```

Testing

```
In [7]: agent.load_weights(fichero_modelo)
        test = agent.test(env, nb_episodes=1000, visualize=False, verbose=0)
        np.savetxt(fichero_test, np.array(test.history["episode_reward"]), delimiter=",")
```

Plots

```
In [8]: train_comparison(models=[modelo, modelo[:-2], "RND"], outfile = "Train"+modelo+".png")
```




```
In [9]: test_comparison(models=[modelo, modelo[:-2], "RND"], outfile = "Test"+modelo+".png")
```